THE CORE KNOWLEDGE SYSTEM

Technical Note No. 426

October 1987

By: Thomas M. Strat and Grahame B. Smith

Artificial Intelligence Center
Computer and Information Sciences Division

Approved for public release; distribution unlimited

Report Documentation Page

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
OCT 1987		**00-10-1987 to 00-10-1987**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
The Core Knowledge System	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
SRI International,333 Ravenswood Avenue,Menlo Park,CA,94025	

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE		**72**	
unclassified	**unclassified**	**unclassified**			

Contents

ii

Chapter 1

Introduction

This document contains an in-depth description of the Core Knowledge System (CKS)—an integrative environment for the many functions that must be performed by sensor-based autonomous and semi-autonomous systems. The CKS itself has been designed to support a wide variety of potential applications. However, special attention has been given to assuring its relevance to a particular application—that of an autonomous land vehicle operating in an unconstrained outdoor environment.

The functionality provided by the system is described, along with discussions of the various design decisions and their associated trade-offs where applicable. This paper is not intended to serve as a user's manual, rather its purpose is to describe the CKS in sufficient detail to allow the reader to ascertain its relevance to a particular application and to provide a technical critique of its strengths and weaknesses.

Chapter 2 contains a complete overview of the goals and architecture of the CKS and the services it provides. It is a slightly revised version of a paper that appeared in the proceedings of the DARPA Image Understanding Workshop held in February 1987 [7]. The remaining chapters examine specific areas in more detail, amplifying important notions and providing examples where appropriate. Chapter 12 describes a scenario that illustrates the envisioned role of the CKS in a complex, sensor-based system. The final chapter gives the current status of the CKS, including its implementation and the directions of ongoing research.

Chapter 2

Architecture of the CKS

2.1 Overview

In this paper, we describe a knowledge system that integrates data from a variety of sensors, as well as from other sources of stored knowledge, to form a world model that is ultimately used by an autonomous system to plan and execute its tasks. The overall architecture of our knowledge system can be viewed as a *community* of interacting *processes*, each of which has its own limited goals and expertise, but all of which cooperate to achieve the higher goals of the system. The various processes may represent sensors, interpreters, controllers, user-interface drivers, planners, or any other information processor that can be imagined. Each process can be both a producer of information and a consumer. Information is shared among processes by allowing them to read data stored by other processes and to update that information. Each process continually and asynchronously updates information based on sensor readings, deductions, renderings, or other interpretations that it makes.

A simple example of object recognition may help to show the system approach. During cross-country navigation an autonomous land vehicle must be able to detect small gullies in its path. Suppose that the vehicle has a range sensor and analyzer that can segment the world into surface terrain and objects on that terrain, a drainage expert that can derive run-off patterns from surface topography, a multi-spectral analyzer that can classify objects on the basis of their spectral signature, a gully detector that determines the likelihood of a gully in the vehicle's path on the basis of the surface drainage pattern and the vegetation cover, and a pilot that makes navigational decisions. Each process contributes expert knowledge about aspects of the world. The determination that there is a gully is not made by a single process that calculates a drainage pattern, vegetation type, and the like, but rather, by each expert looking at the objects that have been placed in the database and deciding if it can contribute further information. When there is sufficient information from which to conclude that a gully is present, the gully expert will annotate the database to this effect. This annotation will then influence steerage decisions

of the pilot process. The decision that there is a gully is not reached by applying a fixed sequence of operations to the data but is made when the available evidence is sufficient to support such a conclusion.

Each process is a *knowledge source* that brings its expertise to the processing of the data that represent the known state of the world. These processes span the range from low-level image processing to symbolic manipulation, and their output will be available for use by all other knowledge sources. The system design is one of expert knowledge sources that know how to draw conclusions based on the available data, but whose knowledge does not necessarily define the actions required to obtain that data. Such procedures are run independently of the processes that use their conclusions. The database that stores these conclusions must be able to accept a vast assortment of data types and make them available to requesting processes.

Our system includes a global database through which information is shared. Because all processes share information, the communication bandwidth between this database and the various processes is of concern. If the granularity of the information to be shared is too fine, then the communication channels will be overloaded with an enormous number of transactions, each of which involves small amounts of data, while a granularity that is too coarse requires complex knowledge sources that are beyond our ability to construct. We view the knowledge sources as substantial entities that attempt to share data objects that are composite in nature. For example, we do not expect that an image-processing routine would write intensity-edge information into the database, but rather that it would share conclusions about the physical objects that are in the world. Of course, these objects may not be identified, nor need they be the final partitioning of the scene into world objects, but they will be entities with which other processes can associate parameters and semantics. This does not mean that the database contains only symbolic objects, but rather that it contains objects that have some semantic character, such as a horizontal planar surface with approximately constant albedo. This minimizes the volume of transactions within the system by associating a significant amount of data with each transaction.

Some knowledge sources may need to communicate with others at a level that is not provided by the global database. Such communication is private to those sources, and implementation is the responsibility of the designers of those processes. This level of information sharing often entails a certain computational speed requirement and usually a processing sequence that can be prespecified. Although any pair of processes may use this form of close coupling, we have focused on expediting the sharing of information that is of a higher level and is substantially unstructured.

If processes are to communicate through the database, the language of communication must be rich enough to allow items to be shared. Relevant information extracted from the database is of little use if the receiving process cannot understand it or use it. With the diversity of information that is available, we choose to share that information through *semantic labels* that classify the information in

the database. These labels must reflect the multiple levels of specificity inherent in the information itself. The labels form a *vocabulary* describing the information that is stored in the database. Accessing information by means of the semantic label allows processes to be independent of the particular data syntax used to store the information. We allow database access through logical combinations of the semantic labels, as well as through procedural definitions passed to the database so that a user may supplement the vocabulary with additional terms. Passing procedural definitions to the database also reduces the communication bandwidth otherwise needed to return the results.

A system that views processes as individual experts that may make conflicting interpretations of the data must have a policy to determine what is stored in the database. For example, if two processes determine that the height of a particular tree is substantially different, which opinion should be stored: the last one given, that of the process with more expertise, or the average of the two? There is no "correct" way to determine a single value. Traditionally, information integration is accomplished as the data are inserted into the database, and the data that are then retained are expected to be free of conflict. In our system, all processes are considered equal, and only their *opinions* are stored. This approach reflects the view that conclusions are not only a function of the data used, but also of the knowledge sources that provide that data, and of the anticipated use of the conclusion. The user of the information should have the opportunity to filter that information with knowledge of both its content and its source. Information in the data store can be modified only by the process that created it, although other processes can cast their opinions.

Any system that consists of a collection of independent, asynchronous processes must have a control mechanism that coordinates these processes to achieve the system's goals. In our design, each process is continually active, going about its task of processing the data that define the current state of the world and placing its opinions in the database. When certain combinations of data occur, we must be able to interrupt particular processes and have them deal with this new information. In our previous example of cross-country navigation, we needed to detect gullies. A clump of snowberry bushes is often an indication of a gully. Consequently, we should activate the gully detector whenever snowberries are detected. A daemon approach is used to implement this strategy. Daemons are placed in the database by the processes that should be informed when particular events occur, and the processes are responsible for determining how to proceed when they are interrupted by these daemons. Control by means of the database is, therefore, data-driven. Alternatively, any process is free to call procedures that are embedded within another process, thus allowing control to be passed by procedure call.

Data-driven control is unlikely to be coordinated to achieve the goals of the system if those goals are not available to the various processes that are performing the data transformations. An important part of sensory integration is planning which

activities will contribute to the more general goals of the larger system in which the sensory system is embedded. In our case, we interface with the goals of a *planning system* that controls the activities of an autonomous vehicle. A planning system is viewed simply as another process or set of processes that may have access to the database. The list of tasks that the vision system is attempting to achieve are data that individual processes must use to establish priorities for their own activities. Conclusions and data transformations, no matter how correct or clever, are irrelevant if they are unrelated to fulfilling the mission of the highest-level system.

2.2 Database

The database that we have designed to store the domain data has many of the usual database features. It stores a collection of *data tokens* that contain the domain information and has a set of indexing structures overlaid on these tokens so that data manipulations based on the domain requirements, such as data retrieval, may be implemented efficiently. Unlike many vision-system databases, the database has a continuity of life that exceeds a single execution of the system. In this respect it is much more like a conventional database, whose integrity and usefulness must persist over an extended period. Data acquired during execution of the system become knowledge stored in the database for future use.

To ensure that the internal integrity of the database is maintained, processes do not have direct access to the data tokens; instead copies of the data are transferred between the database and the process. Clearly, data copying is computationally expensive, which is incompatible with real-time performance. For this reason, a mechanism is provided in the data access language that allows a process to pass a trustworthy procedure to the database so that internal processing can be used to minimize the volume of data transferred and the amount of copying that is necessary. This approach for controlling integrity is dictated by a development environment in which the system is not built by a single person or group but rather is a set of processes provided by disparate implementors. Protecting the data from corruption by an errant process is necessary if we are to avoid rolling back the database to a previous version or editing it between actual uses. However the mechanism used to reduce data copying, sometimes at the expense of jeopardizing integrity, is desirable for certain time-critical processes so that they may achieve real-time performance.

Because the opinions of all processes are stored, the database will contain conflicting and incompatible views of the state of the world. Some processes may exist solely for the purpose of resolving such data inconsistencies. Of course, even these processes will only be allowed to cast an opinion. User processes may choose to take more notice of the opinions of these conflict resolution processes than of the opinions of processes whose conclusions are drawn from less data. The conflict resolution processes will continually process data in the database (as spare computational resources allow), but they are conservative in nature, preferring not to cast an opinion

unless they have overwhelming evidence to support their conclusion. However, a user process may call one of these conflict resolvers to cast an opinion even if it would not have otherwise intervened. Our approach then is to allow inconsistencies to be resolved whenever the data are sufficient to support the resolution, or whenever a user process requires that resolution, i.e. at access time. This approach differs from other approaches that attempt to maintain a consistent data set: in these approaches, resolution must occur at insertion time. The approach we adopt is to resolve when necessary, rather than to resolve always. Often a decision-making process can take action without the need to expend resources in resolving data discrepancies. For example, the navigation module of an autonomous land vehicle may be faced with the conflicting data that the object ahead is either a tree or a telephone pole. If the task is to move forward avoiding obstacles, the vision system does not need to resolve whether the object ahead is a tree or telephone pole. The resolution requirement is a function of the task, not simply the data.

A database that stores opinions will rapidly consume storage resources unless a mechanism is provided that will allow data to be deleted (or at least archived). A process that is the supplier of data may have little ability to evaluate the usefulness of that data, yet it is the useful data that we want available in the database. The approach adopted is to have processes sponsor data; that is, a process (probably a process that uses a particular data token) will allow that data token to be "charged" against its resource allocation. Many processes can sponsor a single data token, and they are charged proportionately. When a process nears its resource limit (or at any time) it can withdraw its sponsorship from any data that it has sponsored. Data that are unsponsored are available for garbage collection (these data may be archived or deleted). In this manner each process is responsible for deciding what data it finds useful, and this collection of data forms the base of current available information. Clearly, this procedure is not fail safe. Critical data may be removed before their criticality is realized. However, the criticality of data is measured in terms of a process's willingness to pay for them and presumably in terms of the current usefulness of those data.

An unsponsored data token will not necessarily be removed immediately. An information producer may not wish to sponsor data for which it has little use, so it may be some time before a sponsor for this information is found. To avoid deleting potentially useful data, the process whose job is to remove data tokens evaluates additional information, such as length of time the token has been in the database, as well as sponsorship information, before it is removed. Data removal is a continuous process, so that the database can be assured of having adequate storage when time-critical tasks demand that computational resources for garbage collection be suspended.

Each process in the system does not have the same resource allocation. At particular times some processes may be more valuable than others. For example, the gully detector used by an autonomous vehicle is a vital process during cross-

country navigation but is rarely used while traveling along roads. One process has the task of allocating database resources to the processes performing useful tasks. The allocation is based on the relative importance of the task and on the frequency with which data tokens are consumed by the process performing that task. Such a frequency measure is a moving statistic that allows the allocation to adapt to the current situation. Data tokens are time stamped to indicate the last time they were modified—that is, the last time a new opinion was added to one of the data slots—and they are time stamped for last use. The time stamps provide data for the resource allocator and the garbage collector.

Data tokens are produced by individual processes and are passed to the database for storage and subsequent retrieval. For the database to access information from within the token, or for a requesting process to be able to extract information from a token, each must either know the form of that information or have some procedure for recovering it. In the design of a system, we can use a standard structure for a data token, such as a record structure in which the position of parameter slots are known, or a standard syntax for the token can be used, such as a list of attribute-value pairs, or we can add functions that retrieve values from the internal data structure of the token, i.e., procedural attachment.

With standard structures, position, rather than name, gives access to the data, but all processes are required to use some predetermined set of structures. In a system in which different processes do entirely different tasks, it is unlikely that we could find, no matter how clever, a small number of data structures that would be natural representations of all the data for all the processes that use those data.

With both fixed syntax and procedural attachment to a data structure, a vocabulary of terms is needed to access the data slots. This is the approach we take. A vocabulary of terms is used that spans the entities and relationships of interest in the application domain. For an autonomous land vehicle, the vocabulary consists of words or labels that describe the outdoor environment, e.g. TREE and HEIGHT, so a process could ask a data token that represents a tree for that tree's height. The actual structure used to hold the data can be invisible to the process that gains access to the information through the labels. The labels must be known by all processes that wish to have access to this information in the database. This semantic level does seem to be the appropriate level on which to share information.

Should we use a fixed syntax like attribute-value pairs to hold the information in the database and provide a simple routine to retrieve the value given the attribute, or should we use the more complex approach of attaching to a data structure a set of functions that can retrieve the value of a data slot given the slot name? We take the latter approach to increase the functionality that is available when a value is retrieved based on slot name. From the point of view of systems building, in which parts of the system are built by independent groups, this approach places the decisions for the form of the data structure and the mechanism for data access within a single group. This approach provides a clean interface with the database. Each
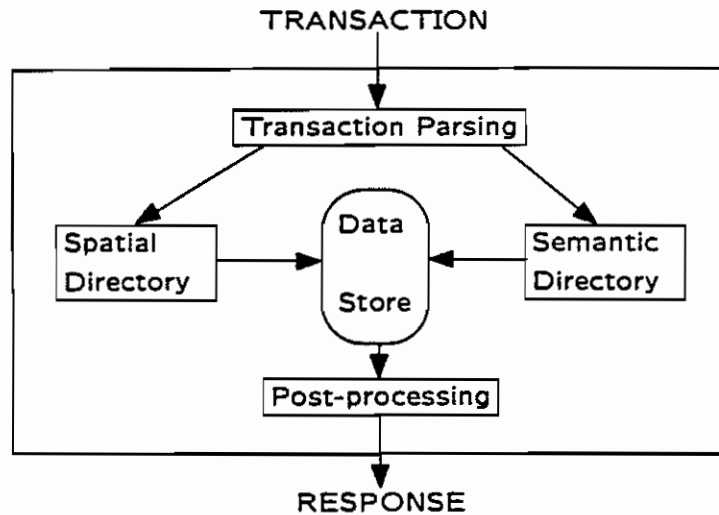
Figure 2.1: Transaction Processing—Autonomous Land Vehicle Database. Access to the data store is by means of a spatial directory, or a semantic directory, or both.

process can select its own internal representations for the data it produces, and those data can be shared through access functions that are based on terms or labels in the vocabulary that describe the underlying domain. A common vocabulary requires that each process know how to translate from its internal representation to information in vocabulary form. This avoids the need for each process to know how to translate into the individual representations used by other processes. Additionally, new processes can be added to the system without retrofitting the new representations to the older processes.

A collection of data tokens is not a database unless there is a means of gaining access to the information in the collection in a manner that does not require a search through the entire set. A set of indexing structures that allows access in a more direct manner must be based on the subsets of the data that need to be retrieved. These structures are, therefore, based on the domain requirements and relate to the semantics of the actual data stored. Our architecture for sensory integration is implemented in the task domain of mobile robot navigation. The indexing structures that we use are associated with the need to retrieve information that is appropriately grouped for the task of navigation in the three-dimensional world. A *spatial directory* that forms subsets of the data based on spatial location, and a *semantic directory* that forms subsets of the data based on object class are the principle indexing schemes that we use to organize storage and retrieval of data tokens. Figure 2.1 gives an overview of transaction processing by means of directories in the database designed to support autonomous vehicle navigation.

2.3 Spatial Directory

The spatial directory organizes the data tokens into groups determined by spatial location. Because an autonomous vehicle may roam about in an extensive environment, we need a representation of that environment that can deal with its spatial extent. In addition, the representation must be efficient in indexing data when the data are distributed nonuniformly over the environment. Data will need to be accessed at various levels of resolution depending on the task that is being addressed. Route planning needs lower-resolution data than does, for example, landmark identification or obstacle avoidance. The world is three-dimensional, but the vehicle is restricted to a two-dimensional surface embedded in this world. Although there are many reasons for choosing a two-dimensional index, such as latitude and longitude, and then representing the third dimension as a data value, we chose to use a three-dimensional index. Our selection was motivated by the advantage such an index gives in encoding spatial relations within the directory, in generating visibility information, and in using this architecture in other spatial domains in which movement is not restricted to a two-dimensional surface.

The three-dimensional index selects a volume in space that we represent as *voxels* [8]. The largest voxel is the world, which is subdivided into smaller volumes as we need to represent spatial position with higher precision. The index granularity is fine enough to be able to position an object in a volume that is precise enough for the application. Recall that this index is an index into a directory; in the directory cell are pointers to the data tokens associated with the volume of space represented by this index. Data tokens need not be placed in the directory at the finest index available but only at the precision with which their spatial location is known. A tree whose position is unknown would be placed in the largest voxel; this being the voxel that represents the entire world.

The voxel-based directory not only gives a range of position resolutions, it also allows different parts of the world to use different resolutions for storing data. In parts of the world that have little associated data, we may choose to place all the pointers to data tokens representing objects in coarse-grained volumes, while the part of the world in which the vehicle is active can be subdivided into finely partitioned volumes. This approach not only provides for multiple resolution access but also allows us to select resolution relevant to the area concerned.

In selecting a voxel-based representation of space, we have the option of dividing that space into regular voxels in which all voxels, at a given level of subdivision of the space, are of equal size, or choosing to divide the space into irregularly sized chunks. Irregularly sized voxels have some attractions; they allow irregularly shaped objects to be confined, and hence indexed, within a volume that matches them. Uniformly sized voxels often are unnecessarily large when they are large enough to contain an irregularly shaped object. However, if irregularly sized voxels are used, multiple indices are needed to allow for overlapping voxels that are indexing

9

different irregularly sized objects in the same volume of space— thus increasing the computational load. We, therefore, use a regular subdivision of space in which each voxel is subdivided into eight equally sized and shaped smaller voxels.

In making this choice, we must address the problem of indexing objects whose shape does not match this partitioning of space. Generally, it is easy to place stationary compact objects within a voxel that can completely contain them, but objects like linear structures, surfaces, and moving objects require alternative approaches. A linear structure like a road, river, telephone wire, or fence is stored as a single data token, but pointers are placed in all the voxels through which the structure passes. The smallest-sized voxels that are appropriate are used; for example, the voxel size for a road will be determined by the road width so that it is assured that the road "fits" within the voxel. The same approach is taken with other extended objects, such as surfaces: a single data token has pointers to it from the set of voxels through which the surface passes.

The size of the voxel is selected by the process inserting the surface into the database, based on such factors as accuracy of the surface shape, and extent. Recall that this placement in space is to aid retrieval, not to specify exactly where things are. Detailed location information is available from within the data token. Usually, a process would place an object in the spatial directory in the smallest voxel possible, although this is not required. When representing terrain patches, for example, placement of the entire patch in a large voxel may be preferable to dividing the patch into more precisely located pieces.

Moving objects are usually compact so they present little problem in placement at their current position, but there may be occasions when it is desirable to represent their track in the directory. The same approach is used as was adopted for linear structures and extended objects: the moving objects are represented by a single data token that is pointed to by the voxels associated with its track.

An advantage of a multiresolution spatial directory is the ease with which approximate location can be represented. An object is placed in a voxel that is large enough to contain the limits of its possible locations. Object location may be approximate because of image processing errors when detecting objects in imagery, or because of lack of knowledge of a sensor's exact position. The latter is particularly relevant in the case of an autonomous vehicle. Data can be added to the database before its position is known, and then, when better location information becomes available, the directory can be updated by moving the data to a smaller volume. If this is not done, the data will be retrieved and examined whenever requests are processed for data from the original larger voxel. A background process whose task is to move each object to its most precise location within the directory (when processing resources are available) accomplishes the directory update and, thereby, achieves retrieval efficiency. Hence, all data can be directly inserted into one directory whether their location is known accurately or only approximately.

Having all data, whose position is known or uncertain within one directory struc-

ture allows us to respond easily to data retrieval requests that seek "all objects that are within a certain volume in space" as well as "all objects that could possibly be within that particular volume of space." Clearly, in the task domain of an autonomous land vehicle, knowing what *might be* ahead and what *is* ahead is necessary for competent navigation and obstacle avoidance. For example, a landmark recognition process needs to know what objects are definitely in some volume, while an obstacle avoidance process is interested in all objects that are possibly in front of the vehicle. Within the voxel structure, "within a volume" maps to the tree of voxels below (finer than) the voxel containing the volume while "possibly within a volume" maps to the tree above (coarser than) the voxel containing the volume.

When data can be retrieved on the basis of their location, then retrievals on the basis of spatial relations are also possible. The spatial directory encodes the spatial relationships between tokens stored in the database. As objects are moved or their spatial positions refined, these spatial relations are maintained without additional processing resources. New objects entered into the database encode their spatial relationship with previously entered data. In our task domain, we expect to retrieve tokens based on relative position—objects to the right of the road, trees casting shadows on the road, and so on. Having an indexing structure that matches the world structure allows this without the overhead that would be presented by alternative schemes, such as a relational database.

The reduction of computational resources used to maintain the database was also instrumental in our treatment of time. The database is always assumed to represent the world at current time. If historical information is to be stored, then it must be time-stamped, otherwise it is implied that the data reflect the state of the world as it currently is. This approach was adopted so that we could avoid elements of the traditional frame problem [1]: if time is a parameter of the data token, then this token has to be updated even when the real data have not changed, but time has passed. We take the usual approach adopted in conventional databases, in that information is assumed to be still true if it has not been altered or specifically marked as applying only to some particular interval of time.

The spatial directory is organized as an *octree* [6] of voxels that span the world. Specifically, a "pointerless" octree that is implemented by multiple hash tables is employed. The use of an octree to implement a voxel representation is natural; the selection of the pointerless approach was based on the expectation that many voxels will contain no data, and many voxels will not be subdivided into smaller units. Hence the more usual approach of using cells with explicit pointers to the finer cells will produce many cells containing mainly null pointers. With the pointerless approach, only voxels that contain data tokens are allocated any storage, and null pointers are not used. Figure 2.2 shows an abstract view (using null pointers) of the way an octree is used to represent the voxel description of the world. The number of levels of hash tables is in fact somewhat less than the number of octree levels, because several octree levels are stored in a single hash table, as shown in Figure
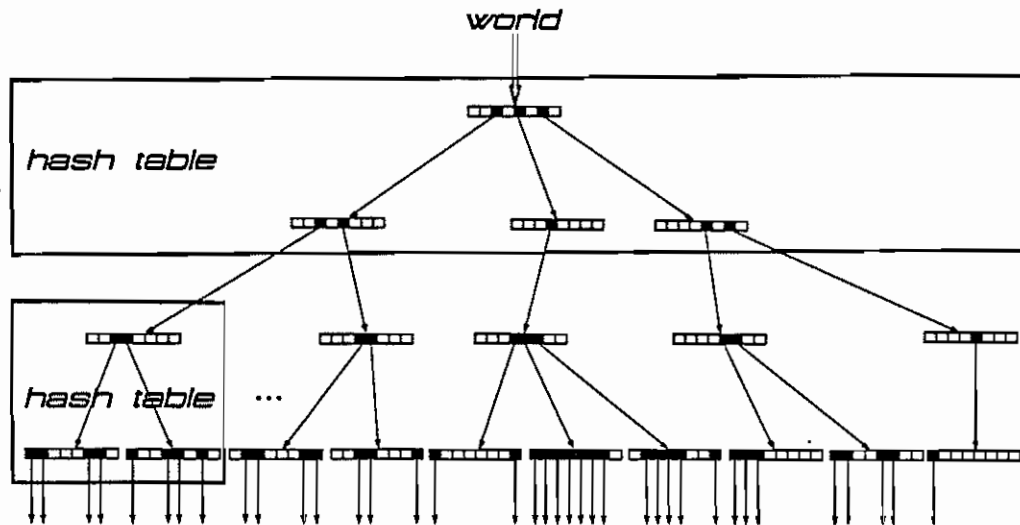
11

Figure 2.2: Octree Representation of the Voxel Description of Space. Hash tables are used to implement the octree; more than one level of the octree is stored in a single hash table.

2.2.

2.4   Semantic Directory

The spatial directory provides an indexing scheme that matches the spatial nature of the data in the task domain; the semantic directory provides an indexing scheme that matches the semantic nature of the data in that domain. As previously mentioned, a vocabulary of terms is used to facilitate communication between processes. The semantic directory specifies these terms and defines the connections among them. The vocabulary provides a set of labels that is used to describe the data tokens in the database. Such a set depends on the task domain, and for autonomous land vehicles we use terms that label objects in the outdoor environment, such as TREE, ROAD, ROCK, MEADOW, or DITCH, as well as terms with less specificity, such as IMMOVABLE-OBJECT, OBSTACLE, or OBJECT.

The need for terms that describe things in the world at various levels of abstraction or multiple levels of resolution is apparent if we wish to interpret imagery as seen from a moving vehicle: objects usually appear first at a distance, at poor resolution, and gradually change form as one approaches them. The needed levels of abstraction are a function of the available processes and their ability to instantiate the terms. There is no point in being able to describe leaves on a tree if the sensors are incapable of resolving objects that small. Equally there is no point in describing trees as belonging to the superset "wooden objects" if no process makes use of that

set. The vocabulary choice that we have made is based on an assessment of the competence of low-level image processing routines and the requirements of higher-level processes. The choice is critical to sensory integration, for within the vocabulary we are restricting the means of integration, the information that higher-level processes can transfer to the low-level routines (and vice versa), and the functionality requirements of both higher and lower-level processes. In absolute terms, successful sensor integration demands selection of an appropriate vocabulary.

Any vocabulary whose constituent terms span a wide range of specificity in a domain will include terms that are related to one another. The second component of the semantic directory, a semantic network [1], defines these connections. The network itself has two parts: one which enumerates the specialization of terms by a graph, that is, a lattice that specifies *subset/superset* relations and that is augmented by the inclusion of the *disjoint set* relation, and a second part that describes the decomposition of composite objects into parts. While the first part indicates relationships that must hold, such as "a pine tree is a tree," the second decomposes composite objects into parts that are usually present, such as "fire engines usually have ladders." The first part of the network is used for inference; for example, in inferring that a PINE-TREE is a TREE, which is an IMMOVABLE-OBJECT, which is an OBJECT, and so on. The second part gives default values that may be used to trigger some process to find them, or may be used by an evidential reasoning process that is attempting, say, to classify an object based on what has been detected and what we might expect to see when viewing that particular object. For example, when a process is attempting to decide whether an object that is composed of several vertical rectangular objects and some horizontal lines could be a portion of a fence, knowledge of the expected parts of a fence is crucial to that determination. Additionally, the network provides a means for inheriting properties from a more general class; e.g. if a tree is usually composed of branches, leaves, and a trunk, then a subclass, like pine trees, will inherit this parts decomposition as its default description. The approach taken reflects, on one hand, the need for the system to reason about objects, while, on the other hand, the system must be able to recognize composite objects on the basis of their likely parts. A mechanism for logical inference and a mechanism for object decomposition that is usual but not unequivocal is therefore provided.

The semantic network is implemented as a graph in which the nodes represent the vocabulary terms and the labeled arcs represent the relationships among terms. Both the subset/superset and disjoint set relations, together with the composite object decomposition, are combined on the one graph using various labels on the arcs to distinguish between them. For example, the lattice fragment

13
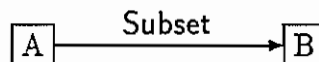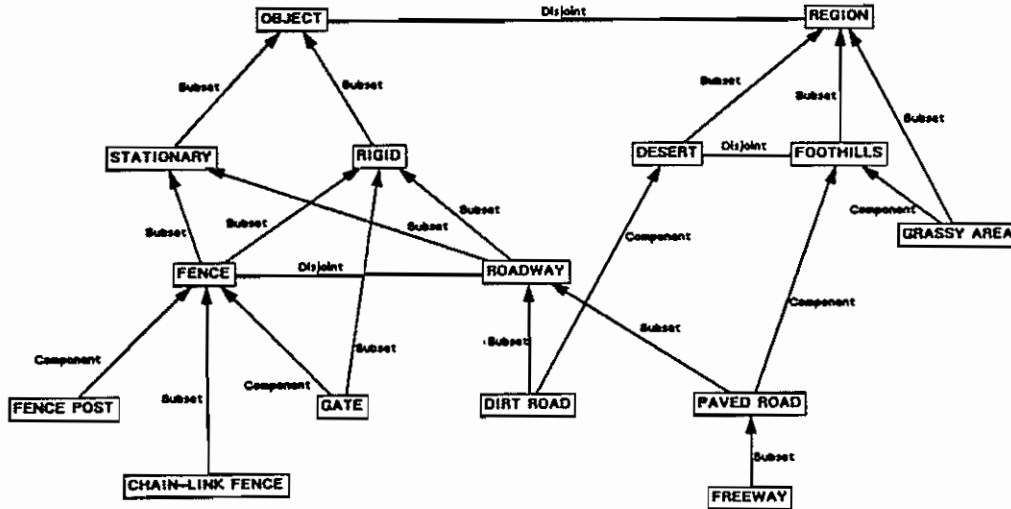
Figure 2.3: Semantic Directory. Implemented as a semantic network, it gives access to the database data tokens by means of their semantic type.

encodes the sentence $\forall x : (A(x) \rightarrow B(x))$, while
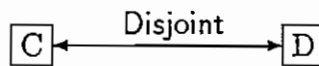
encodes $\neg \exists x : (C(x) \wedge D(x))$.

This network representation allows selected inferences to be made rapidly through graph operations. The particular implementation allows display of the semantic network in its entirety or of selected clusters of related information. Figure 2.3 shows a small part of the semantic network. The graphical display of the network is the interface used to build the semantic directory and to add new words and relations to the vocabulary.

Each node of the semantic network is associated with a vocabulary term, and has pointers to all the data tokens in the database that have been labeled with this term. The nodes of the semantic network can be directly accessed by the vocabulary label, and thus provide a directory to data tokens on the basis of the semantic label. Although we view the semantic directory as a graph structure and display the semantic network as a graph, the implementation uses hash tables for

14

speed of access. When tokens are added to the database or when additional labels are added to a token's description, the semantic directory is updated appropriately.

Data tokens are attached to the most specific network nodes possible. If, for example, a token had been labeled by a process as being a PAVED-ROAD, then it is attached only to the semantic network node for PAVED-ROAD even though all PAVED-ROADs are known to be ROADWAYs. This approach was adopted to save storage as well as to provide a straightforward implementation of the retrieval request to return all objects that are PAVED-ROADs as opposed to all objects that might be PAVED-ROADs. The second descriptor includes objects in the more general class ROADWAY as well as those labeled PAVED-ROAD. Data tokens that represent paved roads are found attached to the nodes of the lattice that form the tree rooted at the node labeled PAVED-ROAD, whereas roadways that *might be* paved are found attached to the nodes of the network tree *above* the node labeled PAVED-ROAD. This arrangement parallels the mechanisms used in the spatial directory to find objects that are at a particular location, as opposed to those that might be at that location. It is the responsibility of the access routines to retrieve the appropriate tokens from the database by means of the semantic network. The ramifications of this approach are described in greater detail in Chapter 7.

The semantic network serves as a partial definition of the meaning of concepts. If a process designer wishes to know what questions he can ask of a data token that represents, for example, a tree, the network specifies the relevant terms, such as HEIGHT, or FOLIAGE-DENSITY. The semantic network defines more than just the communication language between processes; it describes some of the domain knowledge that all processes may use. However, while the concept TREE, for example, may be seen in the semantic network to include PINE-TREEs, and OAK-TREEs, and so on, and while a TREE is an IMMOVABLE-OBJECT and an OBJECT, and while it has properties of HEIGHT, and FOLIAGE-DENSITY, it is not "defined" by the network. The network does not define for a process the concept TREE; it specifies only the concepts that processes can use to communicate about a tree. A particular process may determine that an object is a pine tree on the basis of its temperature and the soil type around it, but it must share its information in terms of the concepts defined in the network. This approach was adopted for important pragmatic reasons—it is impossible to "define" a concept like a tree; yet information about a tree must be communicated in terms that other processes understand.

2.5 Other Directories

The system architecture we have described is independent of the indexing structures that are overlaid on the database; to change those structures requires only changes to the parser that processes database requests (as can be seen in Figure 2.1). The extensibility of the directory system allows future requirements to be accommodated without change to the overall system structure. The spatial and semantic directories

15

were devised to allow an autonomous land vehicle to navigate through a world in which most objects are static and motion comes primarily from the movement of the vehicle itself. In other scenarios, this will be inadequate. In environments in which there are many moving objects, particularly fast moving objects that are likely to affect the mission results, other directories that index the database through additional parameters, such as those associated with movement, are vital. The architecture described has the flexibility to accommodate such extensions.

2.6 Process Control

We have described the various processes that form the system as independent, asynchronous processes that can be activated by means of daemons embedded in the database or by more conventional procedure calls. Each process uses vocabulary terms to interact with the database. Each process is continuously executing, although a process may put itself to sleep only to be awakened when predetermined data conditions exist. Who determines these conditions? Should every process be permitted to determine the conditions needed to interrupt another process? Some processes may be time critical and prefer not to be interrupted. Our approach is to require that the process itself set these conditions within the database. Any process can attach one of its daemons to any data slot of any data token, so that the process will be interrupted whenever any new or changed opinion modifies that data slot. Daemons are attached to data slots rather than data tokens because data tokens usually represent a complex frame and any one process is probably interested in only some aspects of it; for example, the navigational module of an autonomous vehicle will want to be interrupted if a sensor process gives a new opinion on the position of an obstacle, but it is unlikely to need to be interrupted if the obstacle's color changes. It is, therefore, the responsibility of each process to determine when it is to be interrupted. Daemons may also be placed on vocabulary terms or spatial locations to generate an interrupt whenever an object of a particular class or at a particular location is identified.

In a like manner, it is the process that determines what action to take when it is interrupted. The interrupt handler is part of the definition of each process. As processes are quite varied, there is no sense to the notion of a generic interrupt handler. Clearly, a process may choose to continue with what it is doing rather than to process the interrupt if it assesses the current task to be more relevant to mission success than that associated with the interrupt. Conversely, a process may instead suspend or abandon what it is doing in favor of the interrupt. The overall system concept is that of a loosely coupled system in which all processes work on their goals cognizant of the overall mission of the system. Each process determines how it can best support the mission goals and is responsible for the means to achieve this.

The process architecture parallels that of blackboard systems that were brought

to prominence in the building of speech understanding systems [2]. In these systems data are placed on a blackboard; if the combination of data on the blackboard meets the preconditions for a particular procedure to execute, then that procedure is triggered and put on the schedule for computing resources. In an important way, the approach of activating processes using daemons differs from the triggering mechanism used on blackboard systems. There is no pattern matcher whose job is to trigger processes when a particular pattern of data appears in the database (or on the blackboard). For efficiency reasons, the patterns that pattern matchers are to recognize must be predetermined and compiled in at system building time. In a system that is loosely coupled, in which different processes may be present during different executions of the system—in which the system must function even if some of the processes (or hardware) fail—an approach to pattern matching that decentralizes the responsibility for determining whether a process should be triggered seems more manageable. We have chosen to trigger on an opinion being changed rather than on a particular pattern in the data itself. In selecting this mechanism, the cost of the additional processing that is done by the interrupt handler in each process was weighed against the computational cost of running a generalized pattern matcher.

In any system with multiple processes, priority will sometimes need to be given to processes that perform time-critical tasks. At other times, the system could be underused. As a result some processes should be scheduled as foreground jobs, which compete for resources when they request them, while others should be background processes using only spare resources. Some of the background processes have already been identified: the module that resolves data inconsistencies, the one that recovers storage space, and parts of the resource allocator itself. The system should never be idle. Computational resources are allocated to modules by a separate process, a metalevel process, that changes the time slice allocated to various processes. A process that produces data, including opinions, that are used by other processes warrants more resources than a producer of unused data. In addition, a process can request more resources if it determines such a need, so that critical processes can ask for priority.

In the current system implementation all the various processes execute on one computer system, a Symbolics 3600, and all interact through a common virtual address space. This approach was adopted to eliminate the system building necessary to run experiments on multiple processors. However, the conceptual design assumes a virtual environment in which there are many processors running in parallel, with a communications network between them. This accounts for the design decision of the rather loose coupling between processes. On a network of parallel processors, we would expect some processors to be dedicated to particular modules whose computational task is matched to the particular machine hardware. Other processes would be allocated among the available processors. Although we are aware of the bottleneck that might be caused by centralizing the database, we envisage a system

17

in which the process accepting requests for database transactions will be centralized, but the database itself and the procedures that carry out the internal processing may be split across processors.

2.7 Summary

The natural, outdoor environment in which an autonomous land vehicle operates imposes substantial obstacles to the design and successful integration of the vehicle's various sensory, planning, navigational, and control activities. The complexity of the domain and the requirement for high reliability rule out approaches that do not make substantial use of stored knowledge about the environment. An intelligent database that competently contributes to the processes that perform these various system activities is central to the overall design of an autonomous system.

The architecture of the described system is that of a collection of task experts, each implemented as an asynchronous process, that independently update the database with conclusions and deductions about aspects of the world that lie within their domain of competence. Each process must be able to take advantage of relevant knowledge that is available in the database—the knowledge system includes a means for effectively communicating information to the multiple and varied processes that wish to use it. This communication is based on a vocabulary of terms and a set of connections among them that specify the semantics of shared concepts. To support real-time operations, retrieval of data from the database must be supported by database indices that match the semantics of those retrieval requests. Access to the described database is through spatial and semantic directories that organize the various data representations to achieve flexible and timely information retrieval.

18

Chapter 3

Data Tokens

The CKS database stores data tokens that are frame-like objects that are used to store opinions about the properties of entities in the world. A data token is a structure consisting of a collection of data slots in which information is recorded. Each data slot holds opinions (rendered by various processes) of the value (and other data) of the attribute represented by that slot. Each slot has a name that corresponds to a predefined term provided by the CKS or, alternatively, to a user-defined name. For example, if *object-one* is the data token that corresponds to a fence post in the world then the CKS semantic network specifies that data slots named HEIGHT, DIAMETER, and so on, will be part of *object-one*. A user may add any additional slots that are desired, but the existence of these slots will become known to other users only through private conventions.

Every data token always has at least two slots. These are the slots named SEMANTIC-DESCRIPTION and SPATIAL-DESCRIPTION that hold opinions of the semantic and spatial nature of the token and whose specifications are outlined in subsequent chapters. These data slots are used by the CKS to organize its semantic and spatial directories to aid in retrieval. The *type* of a data token, and hence the slots it has, is determined by the CKS vocabulary words listed in the SEMANTIC-DESCRIPTION slot. If a token has a semantic description that lists it as a FENCE-POST, a BROWN object, and a THIN-RAISED-OBJECT, then the data slots present on the token will consist of slots appropriate to these classifications. The data token will be represented as a structure whose type is a composite of the data types that correspond to its various semantic classes. Predefined data types for all CKS semantic classes are present in the system. Additional user-specified data types may be blended with the predefined types. A user can create a data token by making an instance of the appropriate composite data type. As new opinions of the semantic class of an object are added to a token the underlying data type may need to be upgraded. Such an upgrade is provided by the CKS in a manner that is transparent to the user.

It was stated above that each data slot holds opinions (rendered by various

processes) of the value (and other data) of the attribute represented by that slot. The opinions on a data slot are indexed by the processes that cast those opinions. The latest opinion is identified as well. Each opinion consists of three pieces of information; the attribute value, the strength of belief the process has in its opinion, and the time of its belief. Any process can render an opinion about any attribute, and hence any process may have its opinion appended to the opinions of any data slot. However, a process may only have one opinion on any data slot. Should a process offer a further opinion about a data slot for which it already has cast an opinion then that new opinion replaces the old.

As well as the information associated with each opinion there are a number of properties attached to the data token itself. Each token in the database has a unique identifier, time stamps noting when it was last modified and last used, and a list of processes that are "sponsoring" these data. That is a list of processes that are allowing this token to be charged against their resource allocation.

Access to the information in a data token is through the CKS query language, knowledge of the implementation details being largely unnecessary for the database user. Basically, the user can retrieve a whole data token or he can retrieve selected data slots of a token. The user can retrieve all opinions of the value of a data slot, a particular opinion, the latest opinion, and so on. The details of the query language are described in Chapters 6, 7 and 8.

The actual implementation represents data tokens as Lisp flavor instances and the appropriate flavor to represent a particular object can be created by *mixins* of predefined, and user-specified flavor definitions. The predefined flavors, one for each word in the CKS vocabulary, use instance variables to represent the data slots. There are predefined instance variables for all the attributes of an object as specified by the semantic network. Hence the flavor corresponding to the object TREE will have instance variables corresponding to the attributes of a tree, such as HEIGHT, DIAMETER, and so on.

The user who wishes to store a data token in the database corresponding to a *tree* will make an instance of the flavor corresponding to *tree* and pass it to the database through the query language's INSERT-DATA-TOKEN command. If subsequently another process asserts that this object is a TELEPHONE-POLE then the flavor instance will be transparently replaced by a flavor instance that is a *mixin* of the flavors for TREE and TELEPHONE-POLE. A user may extract information from the instance using the attribute names specified for either a TREE or a TELEPHONE-POLE in the semantic network.

The form that the value of a data slot can take ranges from an integer, on the one hand, to an arbitrarily complex data structure on the other. Of course, this value can only be understood by other processes if they know its form. Ordinarily we would expect the form to be implied by the slot, e.g., we would expect the value of the slot for the HEIGHT of a tree to be numeric and in standard units (meters). However, the ability to store a complex data structure allows slots to be used to

20

store data structures that other software tools can use, e.g., a data structure that specifies a model of a scene that is used by a rendering program such as *Supersketch*. The unrestricted form of the values of a data slot supports easy integration of other tools.

Chapter 4

Spatial Specifications

The CKS's spatial representation of the world is volume-based. Objects are considered to be located in volumes of three-dimensional space that generally enclose the object. This enclosing volume has two distinct aspects. The enclosing volume should take into account the size of the object, (i.e. the enclosing volume is a bounding volume), and the enclosing volume should take into account the locational accuracy of the object. The enclosing volume should contain all the space the object may occupy. The CKS does not distinguish between these aspects, objects are associated with enclosing volumes. Of course, this representation of space does not stop a user from placing information within the data token to state position explicitly, or object size etc., but the CKS uses the enclosing volume as its positional information for the purposes of database insertion and retrieval.

At an implementation level the enclosing volumes are mapped to regular three-dimensional space voxels that subdivide the world into regular volumes based on an octree hierarchy of space. An enclosing volume is mapped to the smallest octree voxel large enough to contain the particular volume. This implementation aspect is important as it affects the semantics of retrieval requests. A data token is treated as if the user had placed it in the octree voxel. In practice this means that the retrieval system will err on the side of returning extra tokens that, from exact enclosing volume calculations, would not have been returned.

Volumes may be specified in many different manners and in many different coordinate frames. To store items in the database these volumes must be translated to database volumes in world coordinates. When this translation is not known exactly the transformation must calculate a database volume that is large enough to be an enclosing volume. A volume in space may be represented by its bounding surfaces, a generating function, a set of vertices, and so on. One method, in the current implementation, is to specify a set of three-dimensional cartesian points from which the smallest octree voxel containing all these points is calculated. This allows a volume with planar faces to be specified by its vertices. Within the CKS these enclosing volumes are mapped into octree voxels. The voxels are cubes in

a cartesian coordinate system that is aligned with standard map coordinate axes. For each form of volume specification and for each coordinate system routines are required to carry out this mapping. The CKS will provide standard mappings, a user can supplement this with any particular mappings that he desires by providing a routine that transforms his specifications into one of the known coordinate systems.

The discussion above has made the implicit assumption that we are dealing with compact objects. Extended objects, such as a road, are not compact objects. If we were to associate a road with a long pipelike enclosing volume, it would be mapped within the spatial directory to an octree voxel that would be excessively large and little related to the size and position of the object. To avoid such difficulties a user can specify the enclosing volume as a set of three-dimensional volumes and the object is associated with all these volumes in the database. A spatial specification is a list of volumes that enclose the object. For example, a road is spatially specified by a set of volumes each enclosing a portion of the road—the set of volumes is the enclosing volume of the road. The selection of the size of each volume is decided by the user. In the case of the road, a volume that enclosed its width would seem most appropriate.

In any notation for specifying spatial location there are two properties that need to be selected—the spatial extent that must be specifiable and the level of resolution of that specification. There is a trade-off between these two requirements if the size of the spatial directory is to remain manageable. Within the CKS, we specify this trade-off by a parameter that is the ratio of spatial extent to spatial resolution. We currently have this parameter set to 4096 so that a world comprised of a 4 km × 4 km ground plane is specifiable to the resolution of 1 meter on that ground plane. Of course, the various coordinate systems used and their level of resolution and extent are not restricted by this choice. However, to insert a data token into the database requires a transformation into internal coordinates that has this restriction on extent and resolution. The units of length on each coordinate axis do not need to be the same. However, we choose them to be the same so that vertical as well as horizontal resolution is 1 meter.

The definition of the syntax of a *spatial-description* appears in Appendix A. Within the definition we use the ? character to indicate places where we expect the definitions to be extended with additional specifications. We solicit suggestions of alternatives that should also be included.

Chapter 5

Semantic Specifications

A primary feature of the CKS is a capability for characterizing and retrieving information based upon the semantic content of that information. In this regard, the CKS plays two roles within the context of an autonomous community of processes. First, it provides a common vocabulary so that each process can share information with all other processes without concern for the particulars of their implementations. Second, it contains a knowledge-base that is used to provide a level of understanding of the terms in its vocabulary. Access to this knowledge is gained either implicitly (as when the CKS attempts to evaluate a query) or explicitly (to give user processes access to that knowledge base). The net result is that the CKS can function alternatively as a knowledge base, a database, or as a knowledgeable database in its domain of discourse.

5.1 The Vocabulary

The vocabulary is a set of terms that has been constructed through an examination of the communication requirements of processes operating in the ordinary outdoor environment. The set has been specified as a list of mnemonic labels and appears in Appendix B. It is intended to be the primary means of communication between processes and thus forms their common vocabulary. The meaning of each term is intended to be that which is suggested by the word(s) used; of course, communication can only occur to the extent that the processes using the terms agree on the meaning. We make no attempt to define any of the vocabulary terms completely—such a task is truly impossible. Instead, we stipulate certain relations to hold among these terms, thus creating a partial definition that the agents can use as a basis for communication. Their relation to actual English words strictly serves a mnemonic purpose.

Data tokens are given meaning by a user-process through the assignment of a semantic description. Syntactically, a semantic description is simply an unordered list of vocabulary terms. It is interpreted as meaning that all of the terms represent

true properties of the token. For example, the semantic description (LARGE POST RED) is tantamount to stating that the data token denotes something which is large and is red and is a post. A formal treatment of the semantics of such "semantic-descriptions" is given in Chapter 7. Issues such as the treatment of contradictory vocabulary terms and the implicit implications of terms are discussed there.

The allowable set of semantic descriptions is restricted to include only those terms listed in the vocabulary. The challenge here is not to provide unlimited flexibility in the language, but rather to identify a moderately sized set of properties that are sufficient for interprocess communication. If the properties chosen are not adequate, the vocabulary should be modified; it does not imply a fundamental limitation of the database design. The majority of terms in the vocabulary have been chosen on the basis of their value in fostering communication about sensing and navigation in the natural outdoor environment. Some additional terms have been included for the purpose of exploring the limitations of the representation. It is expected that the vocabulary will evolve as domain requirements become more fully understood. We are particularly interested to learn of semantic concepts that are important for the successful operation of an autonomous land vehicle that have been omitted from the vocabulary.
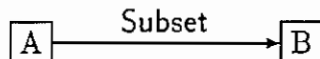
5.2 Attributes

As discussed in Chapter 3, a data token is a framelike structure composed of slots that represent attributes of the denoted entity. Although the user is free to include any attributes he desires, the CKS identifies some special attributes for each class that will be present on all instances of that class. For example, any object identified as being a member of the class ROADWAY, will have the attributes WIDTH, LANES, SPEED-LIMIT, ROAD-MARKINGS, and so on. These special attributes should be used whenever possible so that the information can be uniformly retrieved by other processes that may be unaware of individual representations. The intended meaning of these attributes will be provided in natural-language text to aid the programmer.

The CKS also provides default values for the special attributes. The default will appear as an opinion on the value of that slot along with all other opinions for that slot. A process that reads that slot may alternatively choose to use it exclusively, to combine it with the other opinions, or to ignore it completely. An ordinary inheritance mechanism is provided within CKS to infer appropriate defaults.
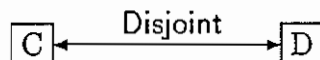
5.3 Semantic Relationships

The semantic relations among vocabulary words are explicitly described by axioms encoded in a semantic network. These formulas are domain-specific and are carefully chosen to support the types of inference necessary for meaningful communication

among independent processes within an autonomous vehicle. They are represented as a collection of machine-readable graph structures. In particular, the semantic network fragment

$$\boxed{A} \xrightarrow{\text{Subset}} \boxed{B}$$

encodes the sentence $(\forall x)[A(x) \rightarrow B(x)]$, while

$$\boxed{C} \xleftarrow{\text{Disjoint}} \boxed{D}$$

encodes $\neg(\exists x)[C(x) \wedge D(x)]$.

These two arcs allow the specification of all possible set relationships. As with the choice of vocabulary terms, the construction of the semantic network is an art that must give careful consideration to the autonomous land vehicle's domain and the anticipated uses of the CKS. As a result, we do not expect that the network in its current form will be the final choice for use on the autonomous land vehicle. Instead, we anticipate a continual process of revision as experiments are conducted with sensory and navigational processes. A graphical reproduction of several slices through the current rendition of the semantic network is given in Appendix C.

A third type of arc is also included in the network. Specifically, the semantic network fragment
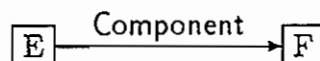
$$\boxed{E} \xrightarrow{\text{Component}} \boxed{F}$$

encodes the knowledge that E typically has F as a component. For example, a car usually has a wheel as a component and a desert typically has cacti. It is important to recognize that we have not chosen to represent absolute component relationships—such strong statements are truly rare in the outdoor world. For example, we would not want to state that all cars have wheels, because a car without wheels is still a car. Instead, the notion of typical containment seems more useful, even if not as clearly defined.

The semantic network represents relations among classes of objects. A relation between an instance and a class is maintained by the semantic directory, which links the data token to the appropriate node in the semantic network. Relations between instances are handled by either of two techniques. If direct access by the name of the relationship is not required, a process may store the information explicitly in the appropriate slot in the data token. For example, a car represented by token201 may have the list (token202 token203 token204 token205) as the value of its COM-

PONENTS slot, where `token202`, `token203`, `token204`, and `token205` are data tokens denoting wheels. For those relations in which direct access may be required, the CKS will employ a relational database to maintain and retrieve the appropriate information. As of this writing, no relational database has been incorporated into the implementation of the CKS.

5.4   Extensions

As a result of repeated experiments with varying processes and environments, it will become necessary to make modifications to the semantic network. The CKS will provide several tools to assist this process.

The primary tool is a network editor that is described in further detail in Chapter 11. It allows us to browse through the semantic network and add, delete or modify graph structure. It is likely to become the primary means for people to maintain the knowledge base.

It will also be necessary to provide user programs with the ability to add vocabulary words to the system. For example, a process with a concept-learning capability may desire to instantiate its newly learned concept. A program that creates a node and links it to the network according to criteria provided will be supplied, but has not yet been written. The major considerations to be resolved here are ensuring the integrity of the network, and imposing resource limitations to prevent adverse effects on performance.

Currently, the network contains three types of arcs. At some point it may be desirable to include additional relations. To do this it will be necessary to write additional code to interpret those arcs. Here again, identification of specific shortcomings must await the results of further experimentation.

Throughout our design, we have endeavored to make the system as extensible as possible. The semantic network will probably be the focus of substantial modification. We actively solicit feedback on the vocabulary and deductive requirements that prospective processes would place upon the CKS.

Chapter 6

The CKS Query Language

The query language that has been developed for the CKS is unusual in several respects because the CKS itself is not simply a special-purpose database. The spatial organization requires a flexible language for describing volumes of space. The fact that it stores both knowledge and data requires a means for accessing either or both. The existence of opinions and apparent contradictions calls for specialized techniques to exploit them. Finally, because the system is designed for use by programmers whose tasks and requirements cannot accurately be predicted, there are provisions for others to extend the query language without interference with each other or existing constructs.

6.1 Transactions

There are six types of transactions supported by the CKS transaction parser. We will describe each of them informally here—further details can be found in succeeding chapters. The syntax for each of the transactions is provided in Appendix A.

(FIND-IDS <query>) This transaction returns a list of data token identifiers that denotes the set of data tokens that satisfy <query>. The various forms of <query> are discussed in Section 6.2. A data token identifier is simply a pointer to a data token—the transaction RETRIEVE-DATA-TOKEN can be used to obtain the data token itself.

(FIND-CLASSES <query>) Sometimes we should like to query the contents of the knowledge base without concern for the state of the database. For example, we might request a list of all trees, and expect to get vocabulary terms like PINE-TREE and OAK-TREE, instead of a list of data tokens denoting particular instances of trees. The FIND-CLASSES transaction accomplishes this. The form of <query> is identical to that used in FIND-IDS. A list of vocabulary terms that satisfy the query is returned.

(INSERT-DATA-TOKEN <data-token>) This transaction is the primary means for putting new information into the CKS. The argument is a data token that has been constructed by the user. The single value returned is the unique identifier for the token.

(RETRIEVE-DATA-TOKEN <id> <slot-process-alist>) This transaction returns the data token associated with <id>. The <slot-process-alist> specifies which opinion is to be selected for each slot before the token is returned.

(INSERT-SLOT-OPINION <id> <slot> <value>) This transaction allows a new opinion to be provided for a particular slot. It is discussed further in Chapter 8.

(RETRIEVE-SLOT-OPINION <id> <slot> <arbitrator>) This transaction returns a value for the slot specified and is also described more fully in Chapter 8. The <arbitrator> determines what method of opinion integration is to be used.

6.2 Queries

In this section, we describe informally the intended meaning of the various forms of query. These constructs compose the means for intensionally representing a set of data tokens or vocabulary words.

As can be seen in Appendix A, queries are either simple or compound. Simple queries are a list of three items: a *qualifier*, a *predicate*, and an *argument*. The argument is a semantic or spatial description or something else, depending upon the identity of the predicate. Each predicate is in reality a three-valued predicate, with "I don't know" being an acceptable value. This allows the user of the CKS to reason appropriately when information is either lacking or inconsistent. Currently four predicates are provided:

IN — This predicate is interpreted to be satisfied only by those data tokens that are contained within the volume denoted by the argument, which must be a spatial description.

IS — This predicate is satisfied by those tokens that belong to the class denoted by the argument, which must be a semantic description.

HAS-AS-PART — This predicate is satisfied by those data tokens of which the argument is believed to be a component.

IS-PART-OF — The data tokens that satisfy this predicate are those that are believed to be components of its argument.

In addition, a new predicate can be defined by providing a LISP function as the predicate in a query. It may take any arguments that are appropriate for it, but must return T, NIL or :MAYBE. For example, a predicate WEIGHS-MORE-THAN could be defined as follows:

```
(defun WEIGHS-MORE-THAN (token ref-weight)
   (let ((weight (RETRIEVE-SLOT-OPINION token :WEIGHT 'LATEST))
      (cond ((null weight) :MAYBE)
            ((> weight ref-weight) T)
            (T NIL))))
```

Programmers who write procedural predicates are cautioned to write them efficiently. While the transaction parser will attempt to optimize the query evaluation, nevertheless it will sometimes be forced to evaluate the predicate on a large list of data tokens.

The qualifier is used to interpret the results of the three-valued predicate's evaluation of each data token in order to construct the list of tokens that are to be returned. Currently, only two qualifiers are acceptable:

APPARENTLY — The list returned contains only those tokens for which the predicate evaluates to T.

POSSIBLY — The list includes all those tokens for which the predicate evaluates to T or :MAYBE.

For the standard predicates, this description indicates the effect of a query, but not the implementation. The actual algorithm used for the standard predicates is much different in order to achieve high performance on very large databases.

The qualifiers, APPARENTLY and POSSIBLY, are also used to discriminate among multiple opinions which may be conflicting or otherwise disparate. Details of their semantics in these cases are provided in the next chapter.

Chapter 7

Logical Interpretation of the CKS Database

The CKS database is a storehouse of the *opinions* of many agents. It is not a database of facts. This design gives it some unusual properties—properties that allow it to function as a central repository of information for a community of processes. It also renders much of the research in database designs inapplicable and has spurred us to develop a new technology for the storage and retrieval of multiple opinions.

The community of processes architecture adopted for the CKS requires that processes be able to communicate their opinions with one another and without undue interference from processes with competing views. The formalization and use of an opinion base in lieu of a database gives rise to the following important features:

- The ability to store information that is *inconsistent.*

- The ability to *integrate* multiple opinions and to allow that integration to depend upon the intended use of the information.

- The ability to *separate* one source's opinions from another's.

These goals have been achieved while retaining an ability to incorporate general knowledge that is universally accepted as being true. An opinion base appears to us to be a better model of how humans store information than a conventional database of facts.

Because of the presence of inconsistent information, first order logic is insufficient for describing the semantics of the database. In what follows, we will resort to a modal logic of belief as a means for interpreting CKS transactions. This logic is extremely expressive and could be used to specify a much richer collection of knowledge. However, we have purposely limited the scope of our logic as a concession to efficiency of implementation. We are, after all, designing the CKS as the core

component of an autonomous vehicle, and implementation issues cannot be ignored. For this reason, the ultimate design is a compromise between the ability to express complicated statements involving beliefs of multiple agents and the ability to retrieve relevant information quickly. Throughout the design, we have been guided by the requirements of autonomous vehicles and have adopted what we feel to be an efficient implementation that does not sacrifice the ability to express the information that must be communicated among the various processes of an autonomous system.

7.1 Semantics

Information is stored in the CKS in the form of data tokens. A *data token* is a framelike object whose internal structure is related to the semantic attributes of the object. Among other things, a data token contains information about the spatial location of an object and some domain-specific properties that are believed to be true about the object. For example, a portion of a data token, token01, might contain:

```
token01
  :SPATIAL-DESCRIPTION   (V1 V2)
  :SEMANTIC-DESCRIPTION (LARGE RED POST)
  :HEIGHT                7.3
     . . .
```

Here the *spatial description* is a list of volume specifications (as described in Appendix A) in which the object is presumed to lie. The *semantic description* is a list of properties that the asserting agent believes to be true about token01.

The precise meaning of CKS transactions is specified with the aid of modal logic. This logic consists of the following components:

Object Constants — The collection of all data tokens and potential data tokens in the database.

Function Constants — None.

Predicates — The set of vocabulary words and the set of possible spatial descriptions.

We adopt the usual syntax for forming WFFs (well-formed formulas) over these symbols as well as all the standard axioms of first order logic. In addition, we assume the existence of implicit axioms that allow us to infer that

- $(\forall x)[V_i(x) \rightarrow V_j(x)]$
  whenever the volume denoted by V_i is completely contained within the volume denoted by V_j, where V_i and V_j are spatial descriptions.

We employ a modal operator B_i to be interpreted as meaning "agent i believes that" so that $B_i[\Phi]$ is interpreted as "Process i believes Φ is true." This operator is similar to that described by Moore [5] and by Konolige [3]. However, our axiomatization is different, as we only develop here those formulas that are needed to interpret the action of the database.

The following axiom schema provides the modal operator with the semantics we desire:

- $\Phi \rightarrow \Psi \Rightarrow B_i[\Phi] \rightarrow B_i[\Psi]$ — The modal operator B_i is closed under deductive inference.

Significantly, this axiomatization does not require that $(B_i[\Phi] \lor B_i[\neg\Phi])$ is true, nor does it require that $(B_i[\Phi] \land B_i[\neg\Phi])$ is false. However, it does require $(B_i[\Phi] \lor \neg B_i[\Phi])$ to be true and $(B_i[\Phi] \land \neg B_i[\Phi])$ to be false. This arrangement allows conflicting beliefs to exist without corrupting the database.

For any proposition Φ, an agent may have any of four states of belief. These are listed below. An example is given of an English statement that would be encoded by each of these states of belief. In the example $\Phi \equiv House(\text{object})$:

- $(B_i[\Phi] \land \neg B_i[\neg\Phi])$ "The object is a house."
- $(B_i[\Phi] \land B_i[\neg\Phi])$ "The object is a house and it's also a car."
- $(\neg B_i[\Phi] \land \neg B_i[\neg\Phi])$ "The object is blue."
- $(\neg B_i[\Phi] \land B_i[\neg\Phi])$ "The object is a car."

7.2 Transactions

Interaction with the CKS is through insertions and queries

7.2.1 Insertions

An insertion is of the form

```
(INSERT-DATA-TOKEN token01)
```

where `token01` is an instance of a data token:

```
token01
  :SPATIAL-DESCRIPTION  (V1 V2)
  :SEMANTIC-DESCRIPTION (LARGE RED POST)
  :HEIGHT               7.3
        . . .
```

Executing (INSERT-DATA-TOKEN token01) has the same effect as asserting

$$B_i[V_1(\texttt{token01})] \wedge B_i[V_2(\texttt{token01})] \wedge$$
$$B_i[LARGE(\texttt{token01})] \wedge B_i[RED(\texttt{token01})] \wedge B_i[POST(\texttt{token01})]$$

Assertions about objects can only be in the form of a conjunction of properties. This approach allows an agent to assert that token02 is a GREEN HOUSE but does not allow him to say, for example, that token02 is either a HOUSE or a BARN. If an agent desires to convey this information, he must rephrase it as, for example, token02 is a BUILDING, with the attendant loss of information. If it is truly important for a process to convey this disjunction precisely, the vocabulary should be extended to include HOUSE-OR-BARN as an acceptable term. It is our expectation that processes will need to communicate only in terms similar to those that have evolved for human communication, and as a result, that there will be little need to resort to such artificial constructs as HOUSE-OR-BARN. The same mechanism can be employed to circumvent the inability to express beliefs about negations.

7.2.2 Queries

The syntax of the query language is provided in Appendix A. The language differs from traditional query languages because the CKS database contains information that is both incomplete and inconsistent. The query language must provide the user with a means for discerning multiple opinions. For this reason, the query language qualifiers APPARENTLY and POSSIBLY are provided to make these distinctions. Loosely speaking, a WFF is true "APPARENTLY" if there is some agent that believes it. It is true "POSSIBLY" if there is some agent who believes it or there is no agent that believes that it is false. These notions will be formalized shortly.

First, we enumerate the situations in which there is only a single agent. Φ is "APPARENTLY" true for the following combinations of beliefs of an agent:

Belief about Φ / Belief about $\neg\Phi$	$B_i[\Phi]$	$\neg B_i[\Phi]$
$\neg B_i[\neg\Phi]$	Yes	No
$B_i[\neg\Phi]$	Yes	No

Similarly, Φ is "POSSIBLY" true for the following combinations of belief:

Belief about Φ / Belief about $\neg\Phi$	$B_i[\Phi]$	$\neg B_i[\Phi]$
$\neg B_i[\neg\Phi]$	Yes	Yes
$B_i[\neg\Phi]$	Yes	No

Before providing exact formulas for the interpretation of CKS queries, it may be useful to examine when Φ is "APPARENTLY" and "POSSIBLY" true when there are two agents involved.

APPARENTLY

Beliefs of Agent 2 / Beliefs of Agent 1	$B_2[\Phi]$ $\wedge$ $\neg B_2[\neg\Phi]$	$B_2[\Phi]$ $\wedge$ $B_2[\neg\Phi]$	$\neg B_2[\Phi]$ $\wedge$ $\neg B_2[\neg\Phi]$	$\neg B_2[\Phi]$ $\wedge$ $B_2[\neg\Phi]$
$B_1[\Phi] \wedge \neg B_1[\neg\Phi]$	Yes	Yes	Yes	Yes
$B_1[\Phi] \wedge B_1[\neg\Phi]$	Yes	Yes	Yes	Yes
$\neg B_1[\Phi] \wedge \neg B_1[\neg\Phi]$	Yes	Yes	No	No
$\neg B_1[\Phi] \wedge B_1[\neg\Phi]$	Yes	Yes	No	No

POSSIBLY

Beliefs of Agent 2 / Beliefs of Agent 1	$B_2[\Phi]$ $\wedge$ $\neg B_2[\neg\Phi]$	$B_2[\Phi]$ $\wedge$ $B_2[\neg\Phi]$	$\neg B_2[\Phi]$ $\wedge$ $\neg B_2[\neg\Phi]$	$\neg B_2[\Phi]$ $\wedge$ $B_2[\neg\Phi]$
$B_1[\Phi] \wedge \neg B_1[\neg\Phi]$	Yes	Yes	Yes	Yes
$B_1[\Phi] \wedge B_1[\neg\Phi]$	Yes	Yes	Yes	Yes
$\neg B_1[\Phi] \wedge \neg B_1[\neg\Phi]$	Yes	Yes	Yes	No
$\neg B_1[\Phi] \wedge B_1[\neg\Phi]$	Yes	Yes	No	No

By extrapolating these results, we are in a position to describe the interpretation of each query in terms of our modal logic.

- (FIND-IDS (APPARENTLY IS W4)) returns a list of those objects in the set

$$\{x \mid (\exists i)B_i[W_4(x)]\}$$

where W4 denotes a vocabulary word. In English, this query corresponds roughly to "Find each data token for which some agent believes W_4 is true of it."

- (FIND-IDS (APPARENTLY IN V3)) returns a list of those objects in the set

$$\{x \mid (\exists i)\mathbf{B}_i[V_3(x)]\}$$

where V3 is a spatial description. This query can be interpreted as "Find each data token such that some agent believes it is contained within the volume V_3."

- (FIND-IDS (POSSIBLY IS W4)) returns a list of those objects in the set

$$\{x \mid [(\exists i)\mathbf{B}_i[W_4(x)] \vee \neg(\exists i)\mathbf{B}_i[\neg W_4(x)]\}$$

This query can be roughly translated as "Find each data token for which some agent believes W_4 could possibly be true." The set of objects which satisfy this query will always include those objects that satisfy the APPARENTLY version.

- (FIND-IDS (POSSIBLY IN V3)) returns a list of those objects in the set

$$\{x \mid [(\exists i)\mathbf{B}_i[V_3(x)] \vee \neg(\exists i)\mathbf{B}_i[\neg V_3(x)]\}$$

This query can be interpreted to mean "Find each data token such that some agent believes that it could possibly be in the volume denoted by V_3."

- (FIND-IDS (AND Query1 Query2))

$$\{x \mid x \in (\text{FIND-IDS Query1}) \wedge x \in (\text{FIND-IDS Query2})\}$$

In English, this can be translated as "Find each data token that satisfies both Query1 and Query2. It can be thought of as the intersection of the sets of tokens that satisfy each query.

- (FIND-IDS (OR Query1 Query2))

$$\{x \mid x \in (\text{FIND-IDS Query1}) \vee x \in (\text{FIND-IDS Query2})\}$$

This can be translated as "Find each data token that satisfies either Query1 or Query2. It can be thought of as the union of the sets of tokens that satisfy each query.

- (FIND-IDS (AND-NOT Query1 Query2))

$$\{x \mid x \in (\text{FIND-IDS Query1}) \wedge x \notin (\text{FIND-IDS Query2})\}$$

This query can be interpreted to mean "Find all data tokens that satisfy Query1 but that do not also satisfy Query2. In set-theoretic terms, it is the set difference of the sets of data tokens that satisfy Query1 and Query2.

In answering negative queries, the database has no Closed World Assumption (i.e., it does not assume that a proposition is false if it cannot be proved true), thus it avoids issues of nonmonotonicity and has no need for circumscription. A negative belief cannot be inserted in the database. However, a negative belief can be deduced using the deductive inference axiom and the general knowledge incorporated in the vocabulary. For example, a process cannot assert that token03 is not a PINE. But if it does assert that the token is an OAK, (i.e., $B_i[OAK(\text{token03})]$), the database will infer that $B_i[\neg PINE(\text{token03})]$.

7.3 Discussion

There are several restrictions upon the statements that a process can make about the world and on the types of queries that can be posed. These restrictions were necessary to enable a practical implementation of the database.

The query language only allows a limited variety of queries. Acceptable queries are limited both by their syntax and by the vocabulary of properties. The query language is not intended to be a universal language. We have designed it so that the only queries that can be posed are those that can usually be retrieved efficiently, given our database architecture. It is important to bear in mind that the limitation of the query language is one that restricts only what questions can be answered efficiently; it does not prevent the identification of data tokens that satisfy an unusual query. When faced with a question that cannot be posed as a syntactically legal query, a user can obtain the exact retrieval by first retrieving a superset of the desired data tokens with an acceptable query, and then examining each token in that set individually for satisfaction of the intended query. Empirical evidence will decide if our designs have been made appropriately.

Chapter 8

Slot Access

The query language described in the previous section provides the means to insert data tokens into the CKS database and to retrieve pointers to those tokens based upon spatial and semantic criteria. In this section, the various mechanisms for gaining access to the information contained within a data token are described.

Data tokens are stored as frames consisting of a number of slots. Externally, each slot has a single value. Internally, however, a separate value is maintained for each process that offers one. When retrieving a slot's value, we specify the combination method desired to combine all values that have been previously provided for that slot. This approach makes it possible to integrate multiple opinions in a manner that is suited to the task at hand. For example, if a robot wants to determine whether its camera will have an unobstructed view over a fence, it should use that opinion that has the greatest value for the HEIGHT slot. On the other hand, if its goal is to keep all the cows in a confined area, it should be interested in the smallest value of HEIGHT.

The following function is used to store a new opinion as the value of a slot:

```
(INSERT-SLOT-OPINION <id> <slot> <value>
                     &optional <auxiliary-data-fields> )
```

INSERT-SLOT-OPINION causes <value> to be stored as the opinion of the calling process for the <slot> of the data token denoted by <id>. If the process has already provided an opinion, it is replaced by <value>. The strength of belief and time of belief may be specified in <auxiliary-data-fields>. If present, this information is stored in the internal representation and can be retrieved with RETRIEVE-SLOT-OPINION. If <slot> is not the name of a currently known slot in <id>, a new slot is created. An error is signalled if <id> is not a valid data token.

Slot values are retrieved from data tokens using the function:

```
(RETRIEVE-SLOT-OPINION <id> <slot> <arbitrator>)
```

RETRIEVE-SLOT-OPINION returns two values. The first can be viewed as the value of <slot> for data token <id>. The particular opinion returned is determined by

`<arbitrator>`, which specifies how multiple opinions are to be integrated by this invocation of `RETRIEVE-SLOT-OPINION`. The second value contains the auxiliary data associated with the returned opinion. `<arbitrator>` can be any of the following:

```
<arbitrator> ::= LATEST  |
                 MIN  |  MAX  |  AVG  |
                 (PROCESS <proc-name>)  |
                 LIST  |  ALIST  |
                 <procedure>
```

Additional arbitrators will be added in the future to support a larger variety of techniques for information integration. Meanwhile, `<procedure>` provides the means for a programmer to make use of a special-purpose procedure.

The functionality of each choice is as follows:

`LATEST` returns the opinion that was most recently provided to the CKS.

`MIN` returns the smallest of all the opinions. `MIN` uses arithmetic comparison if its arguments are numeric, uses alphabetic comparison for arguments that are strings or symbols, and is undefined otherwise.

`MAX` returns the largest of all the opinions.

`AVG` returns the arithmetic average of all the opinions. It ignores any opinion that is nonnumeric.

`(PROCESS <proc-name>)` returns the opinion that was most recently provided by the process denoted by `<proc-name>`. It is undefined if `<proc-name>` has not rendered an opinion.

`LIST` returns a list of all opinions that have been rendered.

`ALIST` returns an alist of all the opinions. Each pair is of the form (`<proc-name>` . `<value>`), where `<proc-name>` is the name of a process, and `<value>` is the most recent opinion provided by that process.

`<procedure>` is the name of a function or lambda expression provided by the programmer. Its single argument is the result that would have been returned if `ALIST` had been used. `<procedure>` should return a value that is to be viewed as the integration of all opinions. For example, a simple implementation of an opinion preference scheme could be implemented by: (`RETRIEVE-SLOT-OPINION <id> 'PRIORITY <slot>`), where `PRIORITY` is defined as

```
(defun PRIORITY (alist)
   (or (cdr (assq Process-1 alist))
       (cdr (assq Process-2 alist))
       (cdar alist)))
```

It is also possible to retrieve an entire data token, given a token <id>.

(RETRIEVE-DATA-TOKEN <id> &optional <slot-process-alist>)

Given a data token <id>, this function returns a flavor instance whose slots are filled with values as determined by the optional argument. By default, each slot receives the most recent opinion expressed by any process (i.e., LATEST). To override the default, include a pair of the form (<slot> <process>) on the <slot-process-alist>.

Chapter 9

Process Design

In Chapter 2, the CKS was described as a community of independent, asynchronous processes. Each process interacts with the database through query language transactions. In the current implementation, a user process is an independent Lisp process that interacts with the CKS through function calls that encode the query language. Each user process consists of all the functions that the user would normally write if that process was to be executed as a stand-alone process plus at most two additional functions. These two additional functions are an *initial function* that the system can use to initialize the process at system start-up, and an *interrupt handler* that allows the CKS to interrupt the user process during execution. Both are relatively straightforward to write as each only needs to deal with aspects of the user process and not the system code.

As far as the user is concerned, his process runs in a Lisp listener window just as if the process was being executed without interaction with the CKS. Except for the effects of the interrupt handler, the user's interaction with the CKS is via query language function calls. The other user processes executing in parallel can be largely ignored. There is one exception to this: because all users are running in a shared environment, and therefore, there may be conflicts in function names, a user should place all his functions in a particular name subspace, i.e. a package, to avoid conflicts with other users.

9.1 Daemons / Interrupt Handlers

A user may place a daemon on any data slot of any data token, or on a vocabulary word in the semantic network, or on a spatial location. This daemon will cause the user process to be interrupted should the data slot of the token be altered in any way, e.g., by changing an opinion or adding a new one. For the other two cases, a vocabulary word or a spatial location, the user process will be interrupted if a new data token is inserted in the database and that token includes the specified vocabulary word in its semantic description or the specified spatial location in its

41

spatial description. When a daemon is triggered by one of the above events the user process is interrupted, the current execution of that process is suspended and the interrupt handler is called. The interrupt handler is free to determine what action to take. It may ignore the interrupt or it may change the course of all future processing. When the interrupt handler returns (if indeed it does return) the suspended execution is resumed.

The daemon mechanism is used to allow a process to setup data-initiated control. Each user process may insert any number of daemons but each of them can only interrupt the process that inserted them. Consequently, the interrupt handler for any user process only needs to know how to handle interrupts that that user process initiated. Writing such an interrupt handler is comparatively easy. Of course, a process that does not insert daemons does not need an interrupt handler.

A user process may associate an expiration condition with each daemon it creates. When the expiration condition is fulfilled the daemon is removed from the database. This feature can be used to eliminate daemons when they are no longer relevant. For example, a daemon on an obstacle might only be relevant while the autonomous vehicle is in the vicinity of that obstacle.

9.2   Initial Functions

Generally each process will have an initial function that is evaluated when the process is created at system start-up. This initial function should never terminate. Typically it will call some initialization routines then place itself in a "sleep" state, or the initialization will start some operation that is continual. If the process is placed in a sleep state then it may have wake-up conditions specified for it or it may rely on CKS daemons to interrupt its sleep.

If a process is already executing at system start-up time there is a mechanism provided for introducing this process to the CKS. Under these circumstances an initial function is not required.

9.3   Example of a User Process

Let us suppose that we want to design a process whose task it is to examine any data token that has been labeled an OBJECT and to determine if the object is a TREE. So let us assume that we have built a function (Is-it-a-tree <data-token-labeled-an-object>) that accepts a data token and places the vocabulary word TREE on the token's semantic description slot if it determines that the object is in fact a tree. Now we build an initial function that:

(a) Places its daemon on the vocabulary word OBJECT in the semantic network.

(b) Puts the process to sleep ("wait forever" so that only an interrupt

42

will revive it).

Next we build an interrupt handler that:

(a) Retrieves from the database the data token that caused the interrupt.

(b) Evaluates (`Is-it-a-tree` <`the-particular-data-token`>).

This user process will be initialized at CKS start-up, placing a daemon on OBJECT and going to sleep. Whenever a new data token is added to the database whose semantic description includes OBJECT, the "*tree-process*" will be awakened and used to determine if the object is a tree.

9.4   A User Process Organized as a Collection of Processes

A user process may wish to organize itself as a collection of independent processes rather than a single process. Support is provided by a function that replaces the usual `make-process` function with one called `make-process-with-database-access`. This sets up the database access for the process. It should be noted that all processes created this way by a particular user process are known to the CKS system by the name of the initiating user process. This implies that daemons placed by any one of the processes will cause all processes in the group to be interrupted, and that interrupt handlers should be built with this in mind. The mechanism allows any process within the user process to insert daemons that interrupt (all) other processes within the group. This mechanism is provided to allow a user to design a task as a collection of processes in which some processes set the daemons to which other processes respond.

43

Chapter 10

System Start-up

During system start-up the user has to make choices about the user processes that are to be active and the initial stored database to be loaded into the CKS. Processes are known to the system at two levels. Any user process that has ever been introduced to the CKS is a *known* process. An *active* process is one that will be initialized during the current execution of the system. The CKS keeps track of all processes that have ever been introduced because the database can be loaded with preset daemons for these processes. Of course, only daemons to active processes will have any effect during the current system execution. During start-up, there are facilities for introducing new user processes and for picking those processes that will be active.

Each process that is active is attached to a display window that gives the user access to that process just as if it were executed independently of all other processes. In addition, there is a window for interacting with the CKS directly that allows a user to interrogate the database directly.

The second aspect of start-up is selecting the database to be loaded. The user can nominate any number of files on which data tokens reside. The identifier attached to each data token is unique across all executions of the system. As a consequence, a user can merge data files associated with different previous executions of the system to form the initial data set for any subsequent execution. There are facilities provided for loading and storing the CKS database as required. The database can be stored in a form that allows the system to be restored, daemons and all, so that multiple executions of the system can simulate a continual execution of the system.

Chapter 11

User Interfaces

The CKS incorporates several interface facilities that enable its users to effectively interact with the system at various levels. Facilities are provided for programmer access as well as human interaction that allow information to be inserted, retrieved, modified, and viewed in a variety of ways. The tools provide access at the level of slots, data tokens, groups of data tokens, as well as at the knowledge level of class attributes, vocabulary words, and relations among classes.

11.1 Query Language

The CKS query language has already been described in Chapter 6. It provides the primary mode of access to the information stored in the CKS. The transaction parser performs some query optimization in an attempt to free the user from concerns of inefficiency. For some special-purpose, time-critical requirements, it may be necessary to provide user access to low-level routines. These routines are available for use by user processes, but such access is discouraged because it side-steps some important functions of the transaction parser, including the enforcement of resource limitations and ensuring the integrity of the data store. Users may query or modify the CKS database at any time by typing transactions into a special interaction window.

11.2 Modeling and Display Facilities

Because the intended domain of the CKS is the physical, outdoor environment, it is sometimes useful to view the contents of the database through synthetic imagery. Several facilities are being provided to construct images of various aspects of the state of the database.

A two-dimensional Sketch Map facility that can construct iconic displays from the symbolic information contained in the database has been implemented. The

choice of icon is determined by information stored in the semantic network, using the standard mechanisms for inheritance of attributes. Particular information, such as the width of a road or the length of a building are used to modify the icon accordingly. A synthetic image is created by evaluating a query to select a set of data tokens and by passing the results of the query to the Sketch Map to be displayed. In this fashion, limited geographic regions can be selected and overlays created based upon particular semantic criteria. An example appears in Figure 11.1.
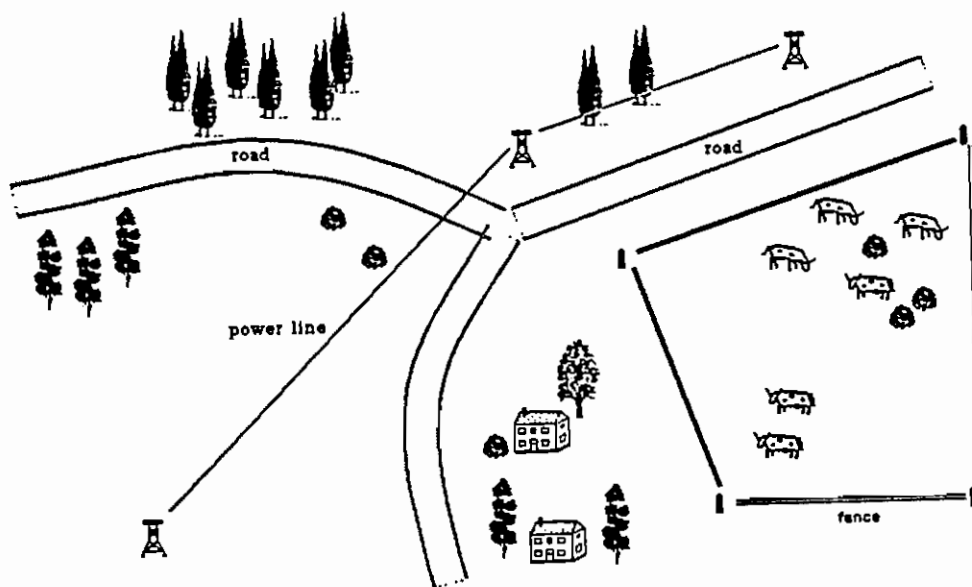
Figure 11.1: A sample display created using the Sketch Map

In addition to providing a viewport into the CKS database, the Sketch Map allows a user to enter new data manually or modify existing information. Any such modifications are maintained internally as the opinions of the user of the Sketch Map. Each icon represents a data token and is mouse-sensitive. Moving the icon across the screen results in a new opinion of its spatial-description. Mousing the icon exposes a menu of the slots of the token and their current values, which can then be changed using the keyboard.

A more powerful Sketch Map facility is being constructed from several tools developed at SRI. This interface will allow full three-dimensional display and modification of any subset of the database. The primary rendering of a landform will be provided by *TerrainCalc* to allow viewing from any perspective; icons corresponding to each data token in the subset of interest will then be superimposed. Some objects will be displayed using *Supersketch* to produce images more realistic than those with stylized icons. Data tokens that have specified geometric modeling in-

formation are displayed using that information; the remainder are displayed using "prototype" models stored in the semantic network. The final result will be a powerful modeling and display system, coupled to a knowledge/database, that is able to provide convincing synthetic imagery from symbolic information stored in the CKS. For more complete details on this modeling and display facility, see [4].

11.3 Knowledge-Base Maintenance Tool

A multifaceted tool has been developed in the course of implementing the CKS that is useful for maintaining the semantic directory. It is constructed on top of *GRASPER II*, a graph representation language and interface developed at SRI. The Knowledge Base Maintenance Tool (KBMT) provides several facilities for viewing and modifying the CKS and its semantic network.

One capability involves menu-driven access to a subset of the query language. This offers retrieval of data according to semantic criteria without requiring the syntax of the query language.

KBMT also allows browsing of the semantic network along several dimensions. The knowledge base itself is fairly large and growing—this tool is indispensable for viewing various slices through the network. For example, we may choose to view information about classes of regions, colors, or parts composition of objects. Alternatively, we may focus on a particular vocabulary word and view all its immediate neighbors. In any of these displays, the nodes are mouseable and additional information is available, including relevant attributes, their default values, and natural language comments to clarify intended meanings. An example appears in Figure 11.2

In addition, KBMT supports maintenance of the semantic network by allowing the user to add, delete, or modify nodes and relations within the network. This can be a dangerous operation with unforeseen consequences and should only be performed with utmost care.

The open-ended architecture together with the common representation of knowledge encourage the development of additional interface facilities. Because all such facilities will make use of a common data representation and means of access, it will be feasible for any user to use new interfaces that are constructed by others. Tools that already exist or are developed independently of the CKS can also be incorporated easily. Because arbitrary information may be stored in data tokens, it is natural to store special-purpose data structures in the database for use by such software packages. This is the mechanism that we have used for employing *Supersketch*, for example, and can be used for other programs as well.
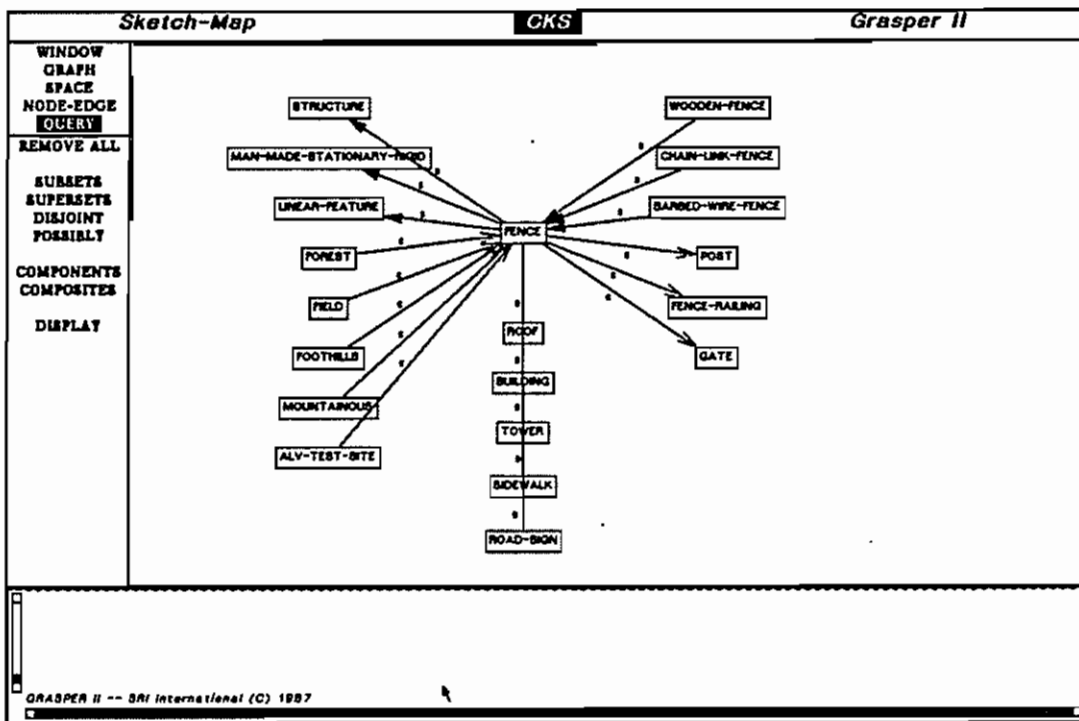
Figure 11.2: Sample display created using KBMT

Chapter 12

Extended Example

As a means of gaining an overview of how the CKS interacts with mission tasks we look at the following example. Suppose that an autonomous land vehicle (ALV) is to carry out the mission of finding a suitable landing site for helicopters in terrain for which reconnaissance data are available. We shall suppose that the reconnaissance data have provided us with topographic information, vegetation cover, feature identification, such as roads and buildings, and so on. Further we shall suppose that this information has been entered into the database using some of the CKS interface tools, such as the two-dimensional map interface.

In setting up the mission, the (human) planner needs to specify what needs to be done and to determine if the reconnaissance data are sufficient to support the ALV task. Firstly, he might want to retrieve from the CKS all the information known about the mission area and render this in the form of an aerial "image." The CKS can retrieve data based on spatial location and can render these data on the basis of the description stored within each data token. Of course, if the specific description is unknown then a default is used to fill in the 'typical' aerial image. Now, the planner may mark an approximate path for the vehicle to take, or he may invoke an automatic planner to find a path from the current location of the vehicle to the approximate area of the landing site. This automatic planner would retrieve terrain data tokens and, using information about the vegetation, determine what areas can be traversed, using topography to determine what areas provide cover from being detected, and using the location of landmarks (maybe pointed out by the human planner) to determine the areas from which the vehicle's sensors can 'see' a landmark and hence support navigation. In making its queries, the planner would integrate multiple opinions in a manner suited to each particular task. For example, in deciding whether a bridge will support the weight of the vehicle, it would request the smallest estimate of the bridge's maximum support weight to ensure the safety of crossing. On the other hand, during obstacle avoidance, the maximum width of a potential obstacle would be of interest. When there is a conflict in opinions the planner may prefer data provided manually during a pre-mission briefing. These

data can be accessed by naming the provider of that information. Once the path is selected, the CKS can supply data related to that path so that a scene renderer can display the view that the ALV's sensors will detect. On viewing these rendered images, a (human) planner can select locations at which the vehicle's sensors need to be reoriented and daemons can be placed in the CKS database so that, when the vehicle is at these locations, the process controlling sensor position will be interrupted.

The selection of a particular path may have been made on the assumptions that particular vegetation cover is available. Daemons can be placed on these data tokens so that if their properties are altered (say, by the object recognizer changing the data-slot FOLIAGE-DENSITY), the automatic planner is called to replan a path.

Once the mission is planned, the ALV starts the task of navigating. Nearby objects are retrieved from the CKS. A process responsible for choosing landmarks would request everything that is "APPARENTLY" in its field of view, so that it would only need to deal with those objects that it should be able to sense. A module whose responsibility is to avoid obstacles would instead request all solid objects that are "POSSIBLY" along the route in order to consider all potential obstacles. The description and properties of these objects can then be used to confirm their presence in the sensor data. The CKS data allows much of the bottom-up object recognition to be replaced by model verification. Of course, discovery processes are still needed to deal with the unexpected and the unknown. This information is usually at a finer resolution than that known *a priori*, and its discovery and inclusion into the database can aid in subsequent excursions in this area by the ALV.

The context provided by these data can be used to select particular routines to process the data. If the road is asphalt (and confirmed by, say, texture analysis) then this information can be used to determine the method for local navigation (say, by finding road edges). Knowing that the road is asphalt can help in calibrating the raw sensor images to help in material identification. The CKS can supply local information, pieced together by many processes to provide constraints for subsequent processing. Should the image analyzer determine that the expected trees to one side of the road are without leaves and, if these trees were assumed to provide cover from detection in the original plan, then the vehicle's planner will be interrupted and used to determine if an alternative route should be used. This interruption will be triggered by the daemon attached to the tree's density data slot that was inserted during planning.

Daemons will also be triggered when the vehicle nears particular locations and the sensors need to be reoriented to detect landmarks.

Once the ALV nears the approximate location of the proposed landing site, the vehicle needs to change modes into one of search. The sensor processes may now be more bottom-up with object recognition modules using the knowledge in the semantic network to determine that the recognition of a post suggests that a fence, or sign post, and so on, may be present and routines should be invoked to look for

these objects in the scene. The required landing site may be described in general terms, flat, no gullies, hidden by the topography from a particular direction, etc. The routine that is looking at the information known about different locations will need to use the semantic network to infer if these conditions are met from the particulars of the data tokens, such as grade of two degrees, grass cover, and the like.

On the return journey the ALV can make use of the data inserted in the CKS during the outward journey. Particularly, it can use the fine resolution data acquired previously to aid in the identification of objects that heretofore have only been seen from a different perspective. Planning could use more up-to-date information, context setting could be based on more complete information, and so on. During debriefing the CKS can render details of the landing site, provide a simulation of the approaches the helicopters may take, and so on.

The CKS plays the role of providing the context in which the task is executed. This may involve setting parameters that a procedure uses, selecting the appropriate procedure, or choosing an appropriate model to be instantiated. The integrating role of the CKS allows knowledge of the domain to be built up incrementally. In the above example on the outward journey, we may have determined some properties of an object, but it may only be on the return journey when we see that object from some different angle that we are able to determine enough information to recognize it. The knowledge embedded in the semantic network allows inferences to be drawn over semantic descriptions. A task that calls for hiding the ALV behind a tree cannot be satisfied by hiding behind a pine tree if the task has no way of determining that pine tree is a subset of tree. The semantic network provides for such inferences. The daemon mechanism provides for data-driven control. Without it a system would be overwhelmed with checking for all types of conditions. The CKS is an integrator of information. Its benefits are realized when there are many processes carrying out interdependent tasks—when one process can determine information that another can use. However, even in the above example where only a few tasks are described, it is natural to divide the problem into a set of modules that have similar functionality to those described. The CKS provides those basic modules and frees the task designer from the need to consider those details. It provides a mechanism for coordinating user-provided modules—again freeing the task designer from the need to consider the coordinating details.

Chapter 13

Status

A first implementation of the CKS has been demonstrated at the DARPA Image Understanding Workshop in Los Angeles in February 1987. As of this writing, all features that are mentioned in this document are included in the implementation. Like any large system, it is expected that CKS will evolve in response to the push and pull of unanticipated uses.

The CKS is not a completely general purpose system—it has been tailored to the domain of spatially oriented tasks. There are many domains that require a system to deal with spatial information and they will be addressed by adding to, rather than changing, the CKS. There are a number of areas in which we are currently working that are intended to provide additional capabilites. These areas include the capture and use of dynamic information, such as is needed to deal with an environment of moving (particularly fast moving) objects; the incorporation of knowledge of the functional nature of activities, for example, the knowledge needed to determine that a particular sequence of activities is part of the description of a higher level of behavior; and implementation of a system for representing a wider range of uncertainty and for reasoning under these circumstances. We expect these developments to expand the functional capabilities of the CKS to support a wide variety of task environments.

The CKS is not a completely general purpose system—it has been tailored to the domain of spatially oriented tasks. There are many domains that require a system to deal with spatial information and they will be addressed by adding to, rather than changing, the CKS. There are a number of areas in which we are currently working that are intended to provide additional capabilites. These areas include the capture and use of dynamic information, such as is needed to deal with an environment of moving (particularly fast moving) objects; the incorporation of knowledge of the functional nature of activities, for example, the knowledge needed to determine that a particular sequence of activities is part of the description of a higher level of behaviour; and implementation of a system for representing a wider range of uncertainty and for reasoning under these circumstances. We expect these

developments to expand the functional capabilities of the CKS to support a wide variety of task environments.

Chapter 14

References

[1] Barr, Avron and Edward A. Feigenbaum, *The Handbook of Artificial Intelligence,* William Kaufmann, Inc., Los Altos, California, 1981.

[2] Erman, Lee D., Frederick Hayes-Roth, Victor R. Lesser, and D. Raj Reddy, The Hearsay-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty, *Computing Surveys,* Vol. 12, pp. 213–253, June 1980.

[3] Konolige, Kurt, "A Deduction Model of Belief and its Logic," SRI Artificial Intelligence Center Technical Note 326, SRI International, Menlo Park, California, August 16, 1984.

[4] Hanson, Andrew J., Alex P. Pentland, and Lynn H. Quam, "Design of a Prototype Interactive Cartographic Display and Analysis Environment," Proceedings of the DARPA Image Understanding Workshop, Vol. II, Los Angeles, pp. 475–482, February 1987.

[5] Moore, Robert C., and Hendrix, Gary G., "Computational Models of Belief and the Semantics of Belief Sentences," SRI Artificial Intelligence Center Technical Note 187, SRI International, Menlo Park, California, 1979.

[6] Samet, Hanan, The Quadtree and Related Hierarchical Data Structures, *Computing Surveys,* Vol. 16, pp. 187–260, June 1984.

[7] Smith, Grahame B., and Strat, Thomas M., "Information Management in a Sensor-based Autonomous System," Proceedings of the DARPA Image Understanding Workshop, Vol. 1, Los Angeles, pp. 170–177, February 1987.

[8] Srihari, Sargur N., Representation of Three-Dimensional Digital Images, *Computing Surveys,* Vol. 13, pp. 399–424, December 1981.

Appendix A

Syntax for the CKS Query Language

```
<transaction>  ::=  (FIND-IDS <query>)  |
                    (FIND-CLASSES <query>)  |
                    (INSERT-DATA-TOKEN <data-token>)  |
                    (RETRIEVE-DATA-TOKEN <id> <slot-process-alist>)  |
                    (INSERT-SLOT-OPINION <id> <slot> <value>)  |
                    (RETRIEVE-SLOT-OPINION <id> <slot> <arbitrator>)

<query>  ::=  <simple-query>  |  <compound-query>

<simple-query> ::=  (<qualifier> IN <spatial-description>)  |
                    (<qualifier> IS <semantic-description>)  |
                    (<qualifier> HAS-AS-PART <semantic-description>)  |
                    (<qualifier> IS-PART-OF <semantic-description>)  |
                    (<qualifier> <3-valued-predicate> [<args> ...] )

<compound-query> ::=   (OR <query> ... <query>)  |
                       (AND <query> ... <query>) |
                       (AND-NOT <query> <query>)


<qualifier> ::=  APPARENTLY  |  POSSIBLY


<semantic-description) ::= (<vocabulary-word> ... <vocabulary-word>)
```

55

```
<arbitrator>  ::= LATEST | MIN | MAX | AVG |
                  (PROCESS <process-name>) |
                  LIST | ALIST |
                  <procedure> |  ?


<spatial-description> ::= ( <spatial-descriptor> ... )

<spatial-descriptor> ::= <volume-by-cartesian-points-in-internal-coordinates>
                       |  ?


<volume-by-cartesian-points-in-internal-coordinates>  ::=
    (:CARTESIAN-POINTS ( <cartesian-vertex-in-internal-coordinates> ... ))


<cartesian-vertex-in-internal-coordinates> ::=
                    ( <cartesian-x-coordinate-in-internal-coordinates>
                      <cartesian-y-coordinate-in-internal-coordinates>
                      <cartesian-z-coordinate-in-internal-coordinates> )

<cartesian-coordinate-in-internal-coordinates> ::= <integer>

    ? = other alternatives to be specified in the future
```

Appendix B

Vocabulary

ABSTRACT-OBJECT
ANIMAL
BARBED-WIRE-FENCE
BLACK
BODY-PARTS
BUILDING
CHAIN-LINK-FENCE
CLOUD
COLD
CURVED
DIRT-ROAD
DUMPSTER
FENCE
FLAT
FOOTHILLS
GRASS-LIKE-TEXTURE
HARD
HIGH-RISE
HILLSIDE
HOT
HUGE
LAKE
LEG
LINEAR-FEATURE
MAN-MADE
MAN-MADE-STATIONARY-FLEXIBLE
MEADOW
MOUND
NARROW
NATURAL-MOVEABLE-RIGID
NEUTRAL-COLOR
OPAQUE
PATH
PLAID
POINT-FEATURE
PUDDLE
RED
RIGID
ROAD-INTERSECTION
ROCK
ROUGH
SELF-PROPELLED
SHINY
SIDEWALK
SMOOTH
SPECKLED
STOP-SIGN
STRIPED
TALL
TEXTURE
TIRE-TRACKS
TREE
TRUCK
VEHICLE
WATER-BODY
WHITE
WINDY
WOODEN-FENCE

ALV
ARCTIC
BARN
BLUE
BRITTLE
BUS
CLIFF
CLUMP-OF-BUSBES
COLOR
DARK
DITCH
DUSTY
FENCE-RAILING
FLEXIBLE
FOREST
GRAY
HAT
HIGHWAY
HILLTOP
HOUSE
ICY-AREA
LANDMARK
LEVEL
LONG
MAN-MADE-MOVEABLE-FLEXIBLE
MAN-MADE-STATIONARY-RIGID
MEDIUM-SIZED
MOUNTAINOUS
NATURAL
NATURAL-STATIONARY-FLEXIBLE
OBJECT
ORANGE
PAVED-ROAD
PLANE
POST
PURPLE
REGION
RIPPLED
ROAD-SIGN
ROCKY
RUT
SHAPE
SHIRT
SLIPPERY
SNOW-COVERED
STATIONARY
STRAIGHT
STRUCTURE
TELEPHONE-POLE
TINY
TOWER
TREE-CROWN
TRUNK
VEST
WATER-TOWER
WIDE
WIRE
YELLOW

ALV-TEST-SITE
ARM
BIRD
BOAT
BROOK
BUSH
CLOTHING
CLUMP-OF-GRASS
COW
DESERT
DOOR
EMBANKMENT
FIELD
FLOWER
GATE
GREEN
HEAD
HILL
HORSE
HUE
JEEP-TRAIL
LARGE
LIGHT
MAMMAL
MAN-MADE-MOVEABLE-RIGID
MATERIAL
METALLIC
MOVEABLE
NATURAL-MOVEABLE-FLEXIBLE
NATURAL-STATIONARY-RIGID
OBSTACLE
PANTS
PERSON
PLANT
POWER-TOWER
RABBIT
RIDGE
RIVER
ROADWAY
ROOF
SANDY
SHED
SHORT
SMALL
SOFT
STONE
STREET
STUMP
TERRAIN
TIRE
TRANSPARENT
TREE-LIKE-TEXTURE
VALLEY
WARM
WEED
WINDOW
WOODEN

Appendix C

Semantic Network

REGION
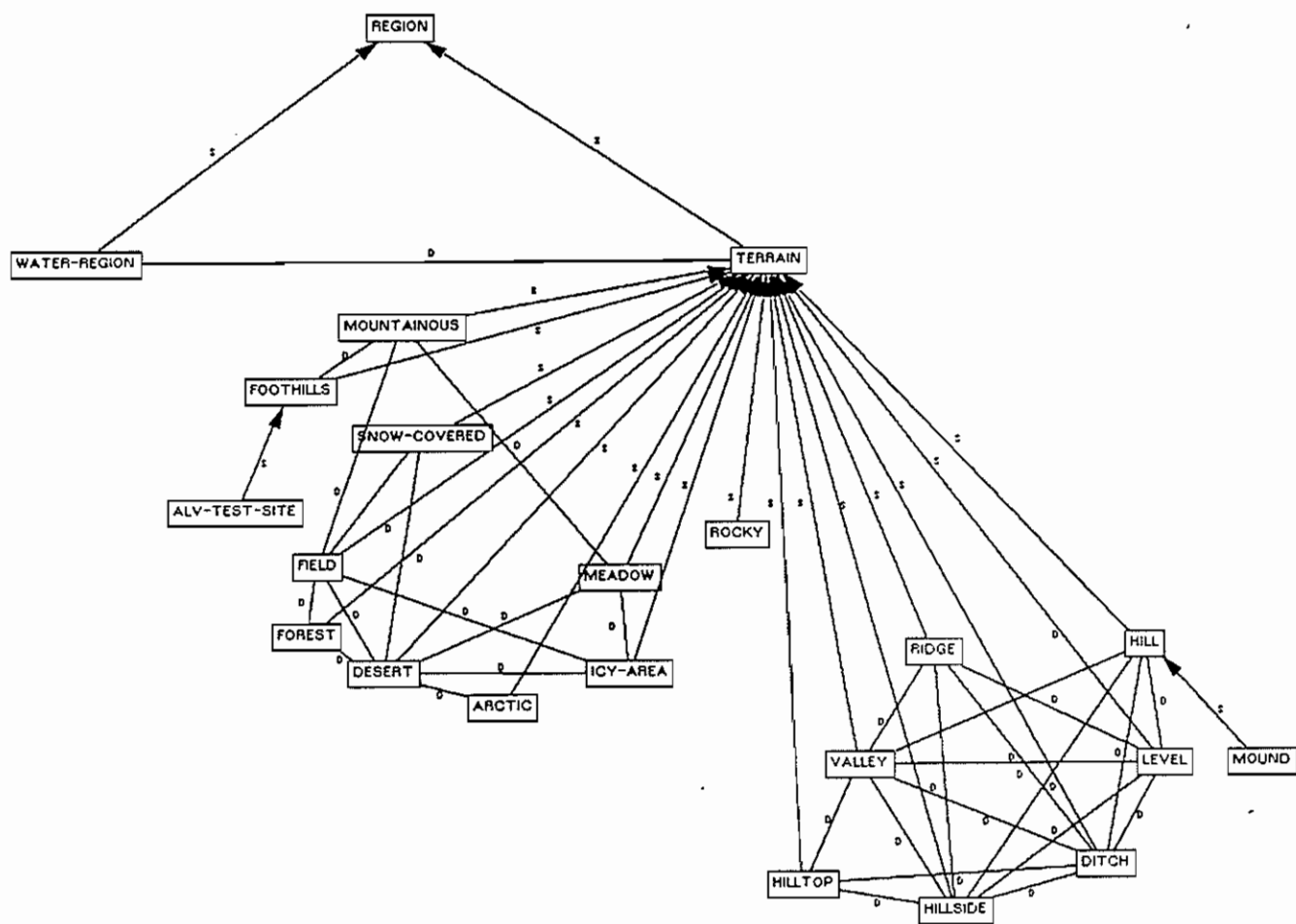
WATER-BODY

ABSTRACT-OBJECT

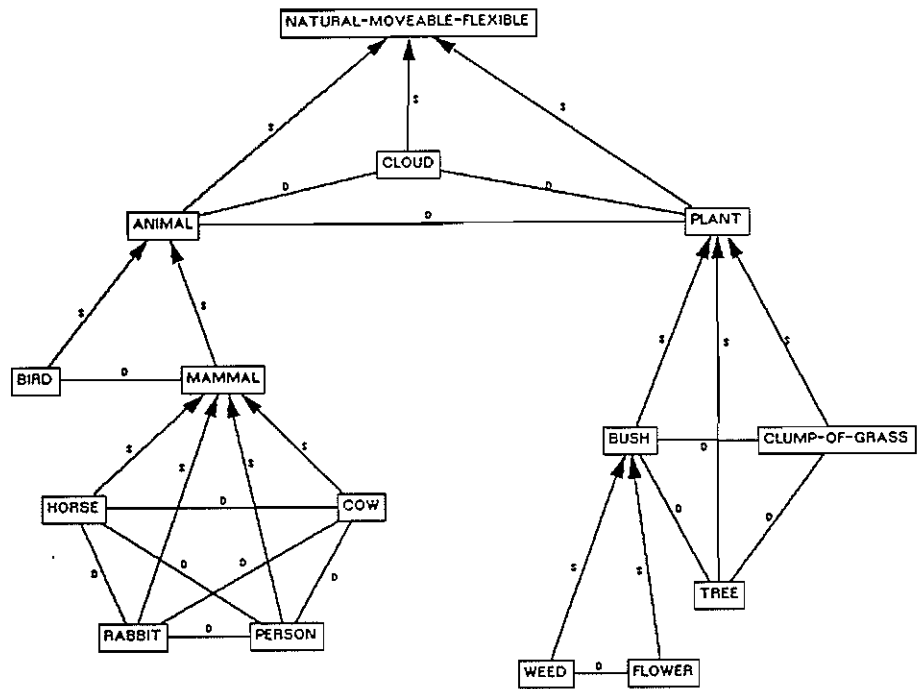61
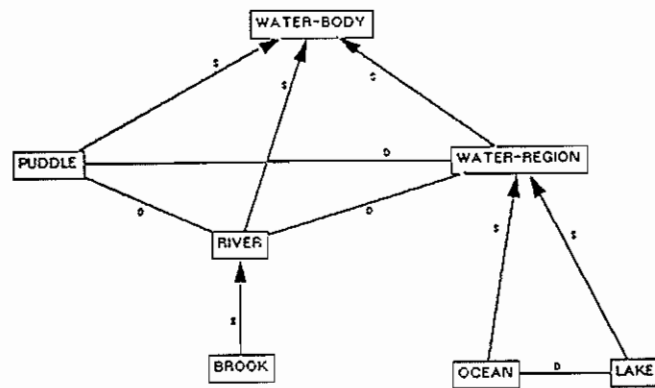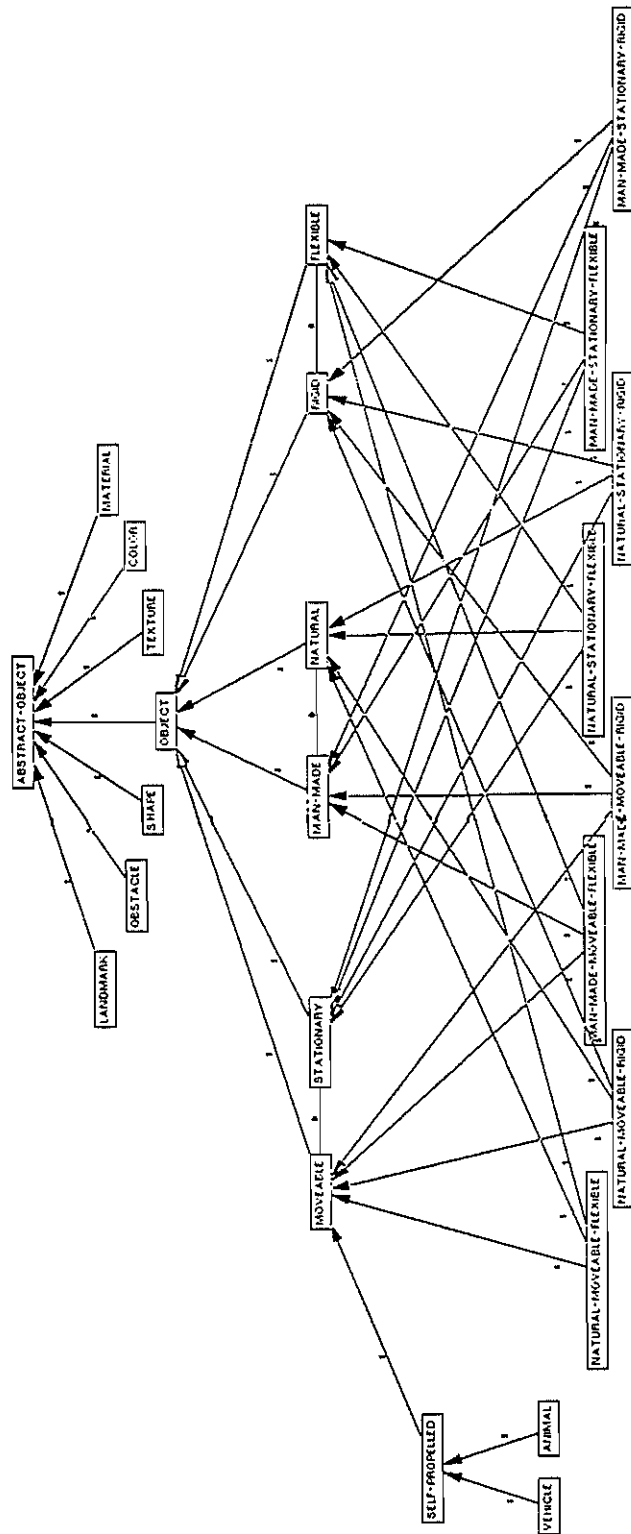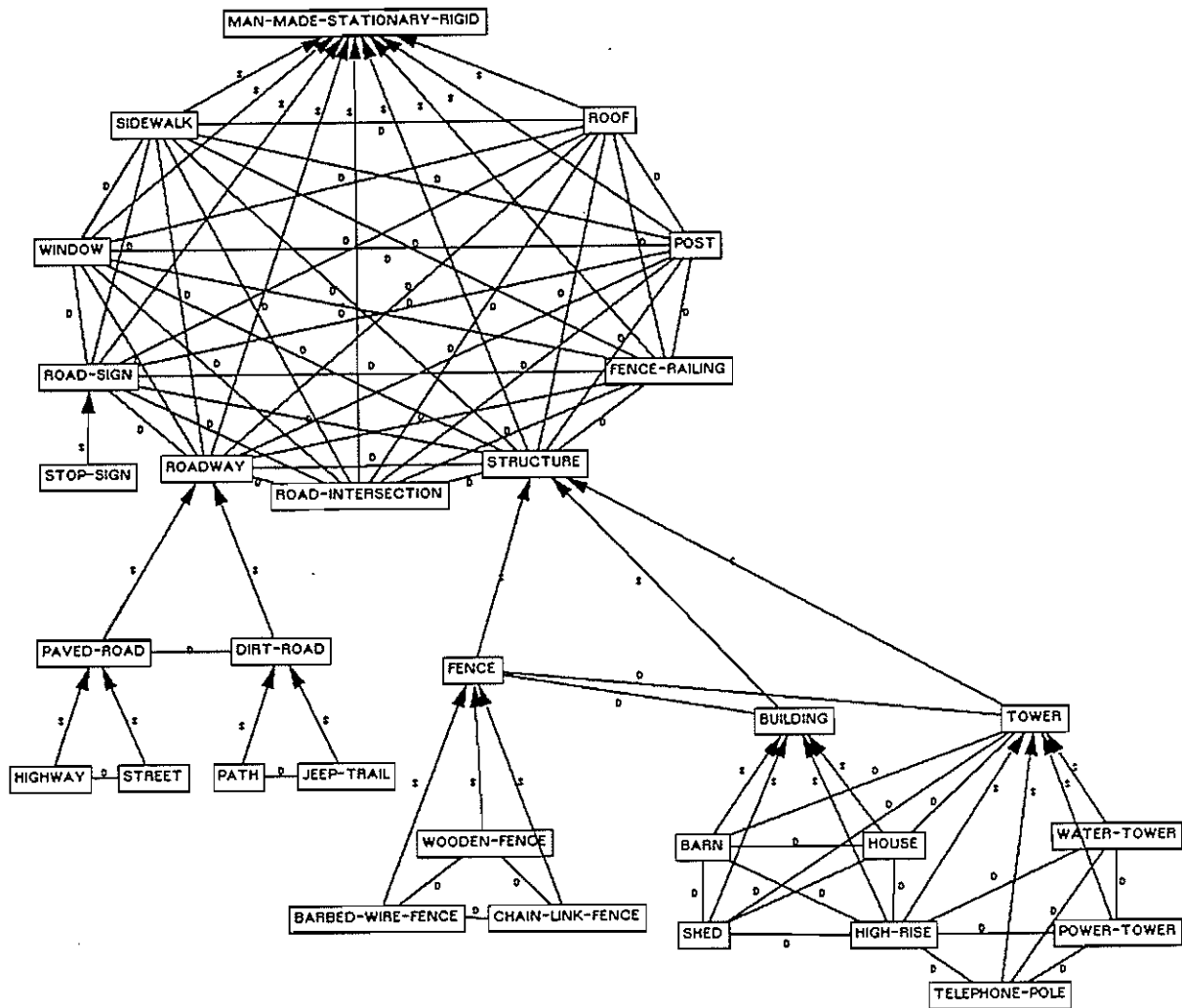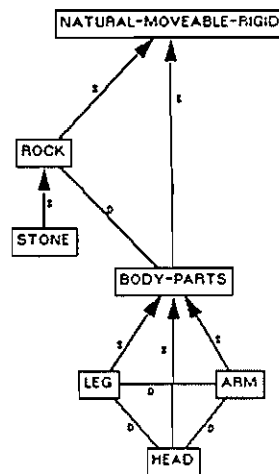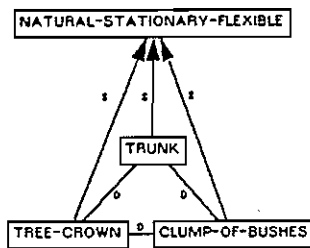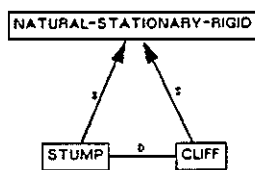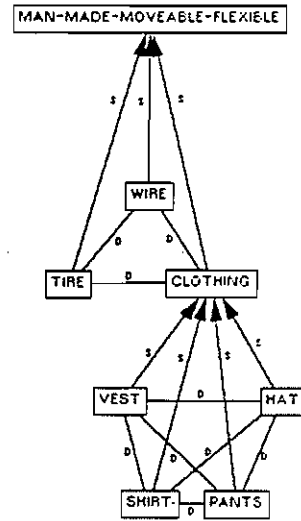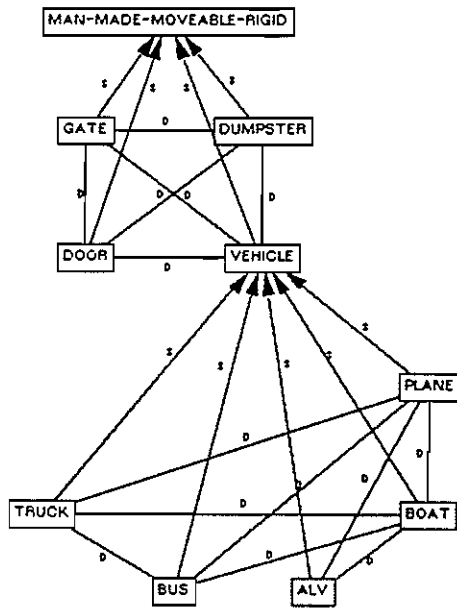
` OBJECT

OBJECT-COMPONENTS

COLOR

COLOR

66

TEXTURE

MATERIAL

SHAPE